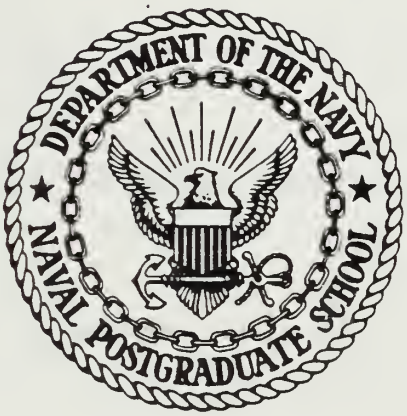
NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

THE DESIGN AND IMPLEMENTATION OF A HIERARCHICAL INTERFACE FOR THE MULTI-LINGUAL DATABASE SYSTEM

by

Timothy P. Benson
and
Gary L. Wentz

June 1985

Thesis Advisor: D. K. Hsiao

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June 1985
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Timothy P. Benson and Gary L. Wentz	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	12. REPORT DATE: June 1985
	13. NUMBER OF PAGES: 204
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Multi-lingual Database System (MLDS), Multi-backend Database System (MBDS), Hierarchical Data Model, Data Language I (DL/I) Attribute=based Data Language (ABDL), Language Interface, Information Management System (IMS)

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Traditionally, the design and implementation of a conventional database system begins with the choice of a data model followed by the specification of a model-based data language. Thus, the database system is restricted to a single data model and a specific data language. An alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach enables the

(Continued)

ABSTRACT (Continued)

user to access and manage a large collection of databases, via several data models and their corresponding data languages, without the aforementioned restriction.
In this thesis, we present the specification and implementation of a hierarchical/DL/I language interface for the MLDS. Specifically, we present the specification and implementation of an interface which translates DL/I language calls into attribute-based data language (ABDL) requests. We describe the software engineering aspects of our implementation and an overview of the four modules which comprise our hierarchical/DL/I language interface.

Approved for Public Release, Distribution Unlimited.

The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System

by

Timothy P. Benson
Captain, United States Marine Corps
B.S., United States Naval Academy, 1978

and

Gary L. Wentz
Captain, United States Marine Corps
B.S., University of Kansas, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

Traditionally, the design and implementation of a conventional database system begins with the selection of a data model, followed by the specification of a model-based data language. An alternative to this traditional approach to database system development is the multi-lingual database system (MLDS). This alternative approach affords the user the ability to access and manage a large collection of databases via several data models and their corresponding data languages.

In this thesis we present the specification and implementation of a hierarchical/DL/I language interface for the MLDS. Specifically, we present the specification and implementation of an interface which translates DL/I data language calls into attribute-based data language (ABDL) requests. We describe the software engineering aspects of our implementation and an overview of the four modules which comprise our DL/I language interface.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. MOTIVATION

During the past twenty years database systems have been designed and implemented using what we refer to as the traditional approach. The first step in the traditional approach involves choosing a data model. Candidate data models include the hierarchical data model, the relational data model, the network data model, the entity-relationship data model, or the attribute-based data model to name a few. The second step specifies a model-based data language, e.g., DL/I for the hierarchical data model, or Daplex for the entity-relationship data model.

A number of database systems have been developed using this methodology. For example, IBM has introduced the Information Management System (IMS) in the sixties, which supports the hierarchical data model and the hierarchical-model-based data language, Data Language I (DL/I). Sperry Univac has introduced the DMS-1100 in the early seventies, which supports the network data model and the network-model-based data language, CODASYL Data Manipulation Language (CODASYL-DML). And more recently, there has been IBM's introduction of the SQL/Data System which supports the relational model and the relational-model-based data

language, Structured English Query Language (SQL). The result of this traditional approach to database system development is a homogeneous database system that restricts the user to a single data model and a specific model-based data language.

An unconventional approach to database system development, referred to as the Multi-lingual Database System (MLDS) [Ref. 1], alleviates the aforementioned restriction. This new system affords the user the ability to access and manage a large collection of databases via several data models and their corresponding data languages. The design goals of the MLDS involve developing a system that is accessible via a hierarchical/DL/I interface, a relational/SQL interface, a network/CODASYL interface, and an entity-relationship/Daplex interface.

There are a number of advantages in developing such a system. Perhaps the most practical of these involves the reusability of database transactions developed on an existing database system. In the MLDS, there is no need for the user to convert a transaction from one data language to another. The MLDS permits the running of database transactions written in different data languages. Therefore, the user does not have to perform either manual or automated translation of existing transactions in order to execute a transaction in the MLDS. The MLDS provides the

same results even if the data language of the transaction originates at a different database system.

A second advantage deals with the economy and effectiveness of hardware upgrade. Frequently, the hardware supporting the database system is upgraded because of technological advancements or system demand. With the traditional approach, this type of hardware upgrade has to be provided for all of the different database systems in use, so that all of the users may experience system performance improvements. This is not the case in the MLDS, where only the upgrade of a single system is necessary. In the MLDS, the benefits of a hardware upgrade are uniformly distributed across all users, despite their use of different models and data languages.

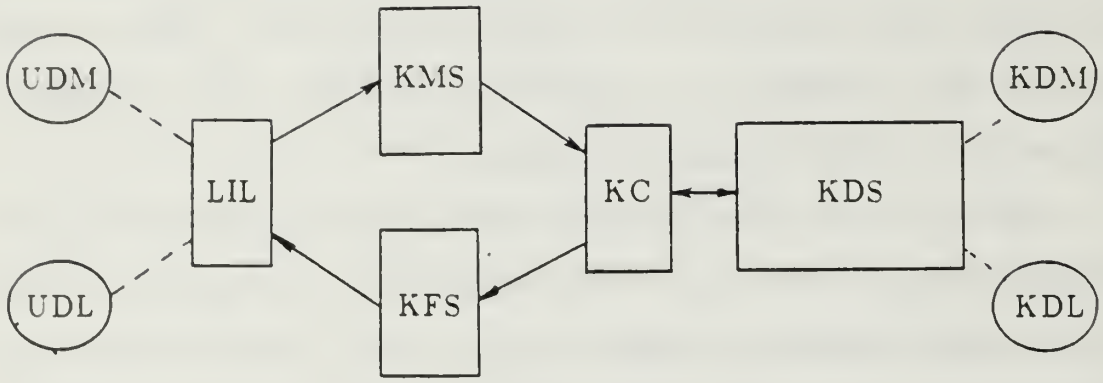
Thirdly, a multi-lingual database system allows users to explore the desirable features of the different data models and then use these to better support their applications. This is possible because the MLDS supports a variety of databases structured in any of the well-known data models.

It is apparent that there exists ample motivation to develop a multi-lingual database system with many data model/data language interfaces. In this thesis, we are developing a hierarchical/DL/I language interface for the MLDS. We are extending the work of Banerjee [Ref. 2] and Weishar [Ref. 3], who have shown the feasibility of this particular interface in a MLDS.

B. THE MULTI-LINGUAL DATABASE SYSTEM

A detailed discussion of each of the components of the MLDS is provided in subsequent chapters. In this section we provide an overview of the organization of the MLDS. This assists the reader in understanding how the different components of the MLDS are related.

Figure 1 shows the system structure of a multi-lingual database system. The user interacts with the system through the language interface layer (LIL), using a chosen user data model (UDM) to issue transactions written in a corresponding model-based user data language (UDL). The LIL routes the user transactions to the kernel mapping system (KMS). The KMS performs one of two possible tasks. First, the KMS transforms a UDM-based database definition to a database definition of the kernel data model (KDM), when the user specifies that a new database is to be created. When the user specifies that a UDL transaction is to be executed, the KMS translates the UDL transaction to a transaction in the kernel data language (KDL). In the first task, the KMS forwards the KDM data definition to the kernel controller (KC). The KC, in turn, sends the KDM database definition to the kernel database system (KDS). When the KDS is finished with processing the KDM database definition, it informs the KC. The KC then notifies the user, via the LIL, that the database definition has been processed and that loading of the database records may begin. In the second task, the KMS



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KC : Kernel Controller
KFS : Kernel Formatting System
KDM : Kernel Data Model
KDL : Kernel Data Language
KDS : Kernel Database System

Figure 1. The Multi-Lingual Database System.

sends the KDL transactions to the KC. When the KC receives the KDL transactions, it forwards them to the KDS for execution. Upon completion, the KDS sends the results in KDM form back to the KC. The KC routes the results to the kernel formatting system (KFS). The KFS reformats the results from KDM form to UDM form. The KFS then displays the results in the correct UDM form via the LIL.

The four modules, LIL, KMS, KC, and KFS, are collectively known as the language interface. Four similar

modules are required for each other language interface of the MLDS. For example, there are four sets of these modules where one set is for the hierarchical/DL/I language interface, one for the relational/SQL language interface, one for the network/CODASYL language interface, and one for the entity-relationship/Daplex language interface. However, if the user writes the transaction in the native mode (i.e., in KDL), there is no need for an interface.

In our implementation of the hierarchical/DL/I language interface, we develop the code for the four modules. However, we do not integrate these modules with the KDS as shown in Figure 1. The Laboratory of Database Systems Research at the Naval Postgraduate School is in the process of procuring new computer equipment for the KDS. When the equipment is installed, the KDS is to be ported over to the new equipment. The MLDS software is then to be integrated with the KDS. Although not a very difficult undertaking, it may be time-consuming.

C. THE KERNEL DATA MODEL AND LANGUAGE

The choice of a kernel data model and a kernel data language is the key decision in the development of a multi-lingual database system. The overriding question, when making such a choice, is whether the kernel data model and kernel data language is capable of supporting the required

data-model transformations and data-language translations for the language interfaces.

The attribute-based data model proposed by Hsiao [Ref. 4], extended by Wong [Ref. 5], and studied by Rothnie [Ref. 6], along with the attribute-based data language (ABDL), defined by Banerjee [Ref. 7], have been shown to be acceptable candidates for the kernel data model and kernel data language, respectively.

Why is the determination of a kernel data model and kernel data language so important for a MLDS? No matter how multi-lingual the MLDS may be, if the underlying database system (i.e., KDS) is slow and inefficient, then the interfaces may be rendered useless and untimely. Hence, it is important that the kernel data model and kernel language be supported by a high-performance and great-capacity database system. Currently, only the attribute-based data model and the attribute-based data language are supported by such a system. This system is the multi-backend database system (MBDS) [Ref. 1].

D. THE MULTI-BACKEND DATABASE SYSTEM

The multi-backend database system (MBDS) has been designed to overcome the performance problems and upgrade issues related to the traditional approach of database system design. This goal is realized through the utilization of multiple backends connected in a parallel

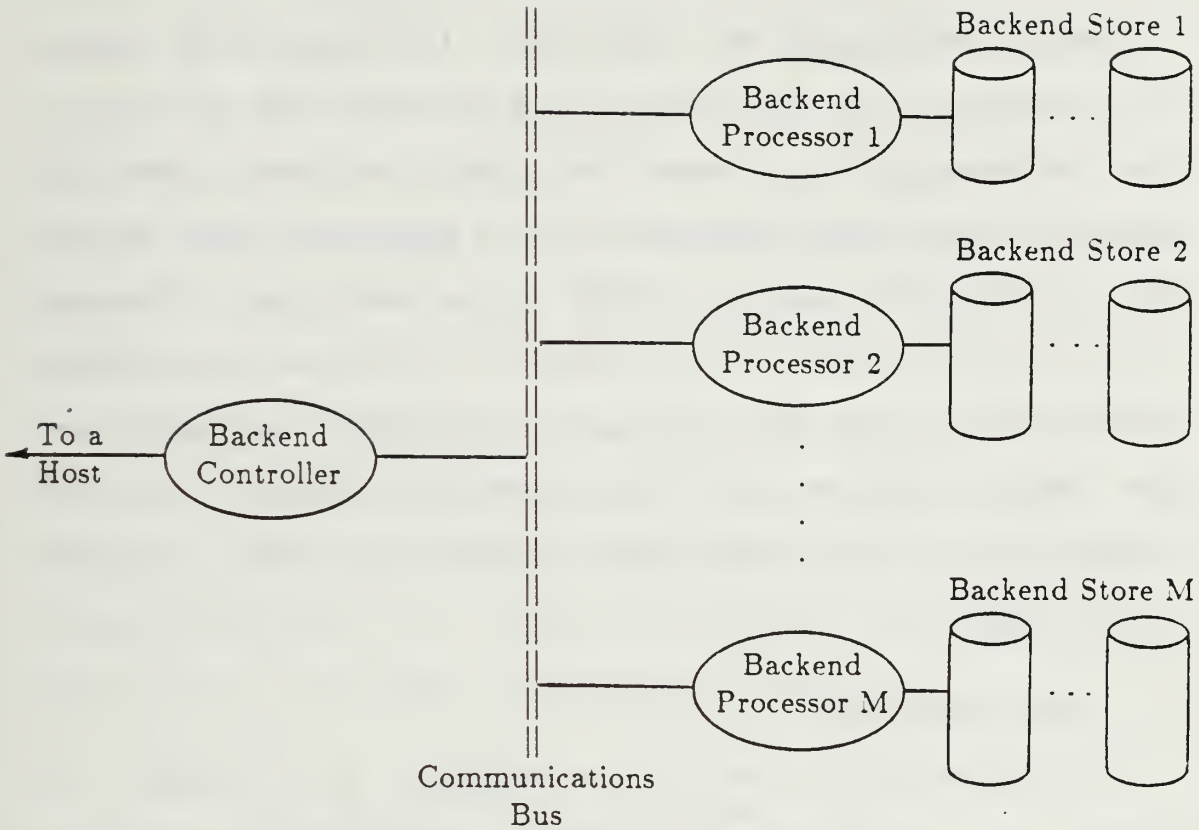


Figure 2. The Multi-Backend Database System.

fashion. These backends have identical hardware, replicated software, and their own disk systems. In a multiple backend-configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk system of the backends. The controller and backends are connected by a communication

bus. Users access the system through either the hosts or the controller directly (see Figure 2).

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a reciprocal decrease in the response times for the user transactions when the number of backends is increased. On the other hand, if the number of backends is increased proportionally with the increase in databases and responses, then MBDS produces invariant response times for the same transactions. A more detailed discussion of MBDS is found in [Ref. 8].

E. THESIS OVERVIEW

The organization of our thesis is as follows: In Chapter II, we discuss the software engineering aspects of our implementation. This includes a discussion of our design approach, as well as a review of the global data structures used for the implementation. In Chapter III, we outline the functionality of the language interface layer. In Chapter IV, we articulate the processes constituting the kernel mapping system. In Chapter V, we provide an overview of the kernel controller. In Chapter VI, we describe the kernel formatting system. In Chapter VII, we conclude the thesis.

Appendix A covers the data structure diagrams for the shared and local data. The detailed specifications of the interface modules (i.e., LIL, KMS, KC, and KFS) are given in Appendices B, C, D, and E, respectively. Appendix F is a users' manual for the system. The specifications of the source data language, DL/I, and the target data language, ABDL, is found in [Ref. 17: pp. 282-307] and [Ref. 7], respectively.

Throughout this thesis, we provide examples of DL/I requests and their translated ABDL equivalents. All examples involving database operations presented in this thesis are based on the education database described in Date [Ref. 17: pp. 279-284], and shown in Figure 3.

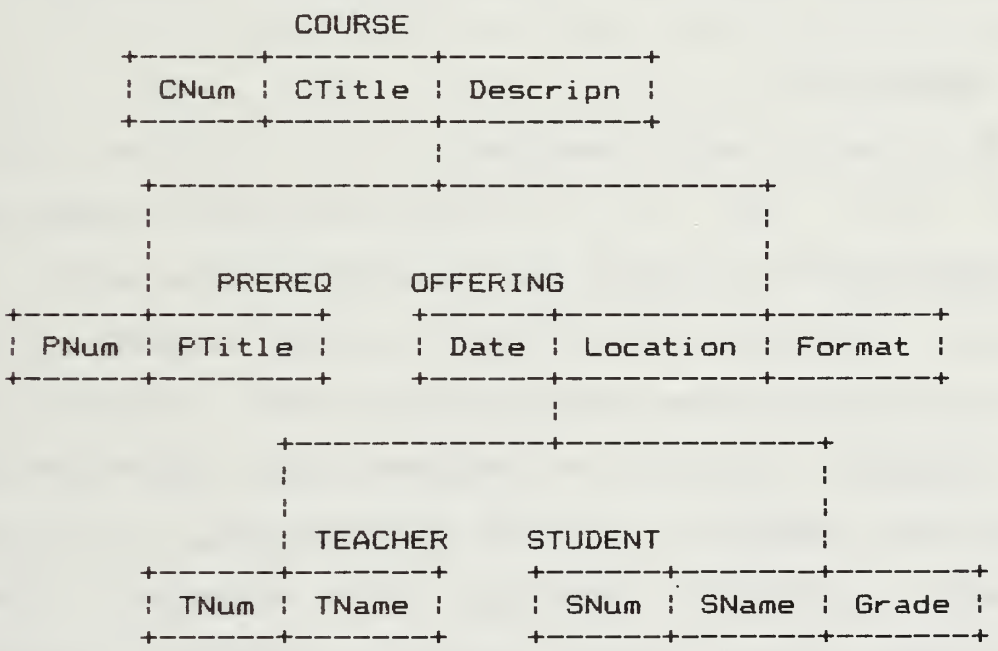


Figure 3. The Education Database.

II. SOFTWARE ENGINEERING OF A LANGUAGE INTERFACE

In this chapter, we discuss the various software engineering aspects of developing a language interface. First, we describe our design goals. Second, we outline the design approach that we have taken to implement the interface. Included in this section are discussions of our implementation strategy, our software development techniques, and salient characteristics of the language interface software. Then, we provide a critique of our implementation. Fourth, we describe the data structures used in the interface. And finally, we provide an organizational description of the next four chapters.

A. DESIGN GOALS

We are motivated to implement a DL/I interface for a MLDS using MBDS as the kernel database system, the attribute-based data model as the kernel data model, and ABDL as the kernel data language. It is important to note that we do not propose changes to the kernel database system or language. Instead, our implementation resides entirely in the host computer. All user transactions in DL/I are processed in the DL/I interface. MBDS continues to receive and process requests in the syntax and semantics of ABDL.

In addition, we intend to make our interface transparent to the user. For example, an employee in a corporate environment with previous experience with DL/I could log into our system, issue a DL/I request and receive resultant data in a hierarchical format, i.e., a segment. The employee requires no training in ABDL or MBDS procedures prior to utilizing the system.

B. AN APPROACH TO THE DESIGN

1. The Implementation Strategy

There are a number of different strategies we might have employed in the implementation of the DL/I language interface. For example, there are the build-it-twice full-prototype approach, the level-by-level top-down approach, the incremental development approach, and the advancemanship approach [Ref. 10: pp. 41-46]. We have predicated our choice on minimizing the "software crisis" as described by Boehm [Ref. 10: pp. 14-31].

The strategy we have decided upon is the level-by-level top-down approach. Our choice is based on, first, a time constraint. The interface has to be developed within a specified time, specifically, by the time we graduate. And second, this approach lends itself to the natural evolution of the interface. The system is initially thought of as a "black box" (see Figure 1) that accepts DL/I transactions and then returns the appropriate results. The "black box"

is then decomposed into its four modules (i.e., LIL, KMS, KC, and KFS). These modules, in turn, are further decomposed into the necessary functions and procedures to accomplish the appropriate tasks.

2. Techniques for Software Development

In order to achieve our design goals, it is important to employ effective software engineering techniques during all phases of the software development life-cycle. These phases, as defined by Ledthrum [Ref. 11: p. 27], are as follows:

(1) Requirements Specification - This phase involves stating the purpose of the software: "what" is to be done, not "how" it is to be done.

(2) Design - During this phase an algorithm is devised to carry out the specification produced in the previous phase. That is, "how" to implement the system is specified during this phase.

(3) Coding - In this phase, the design is translated into a programming language.

(4) Validation - During this phase, it is ensured that the developed system functions as originally intended. That is, it is verified that the system actually does what it is supposed to do.

The first phase of the life-cycle has already been performed. The research done by Demurjian and Hsiao [Ref. 1] has described the motivation, goals, and structure of the MLDS. The research conducted by Weishar [Ref. 3] has extended this work to describe in detail the purpose of

the DL/I interface. Hence, the requirements specification is derived from the above research.

We have developed the design of the system using the above specification. A Systems Specification Language (SSL) [Ref. 12] is used extensively during this phase. The SSL has permitted us to approach the design from a very high-level, abstract perspective by:

(1) enhancing communications among the program team members,

(2) reducing dependence on any one individual, and,

(3) producing complete and accurate documentation of the design.

Furthermore, the SSL has allowed us to make an easy transition from the design phase to the coding phase.

We have used the C programming language [Ref. 13] to translate the design into executable code. Initially, we were not conversant in the language. However, our background in Pascal and the simple syntax of C, have made it easy for us to learn. The biggest advantage of using C is in the programming environment that it resides (i.e., the UNIX operating system). This environment has permitted us to partition the DL/I interface, and then manage these parts in an effective and efficient manner. Perhaps, the only disadvantage with using C is the poor error diagnostics, having made debugging difficult. There is an on-line debugger available for use with C, in UNIX, for debugging.

We have avoided this option, and instead, used conditional compilation and diagnostic print statements to aid in the debugging process. To validate our system we have used a traditional testing technique; path testing [Ref. 14]. We have checked boundary cases, such as the single and multiple DL/I ISRT operations. And we have tested those cases considered "normal". It is noteworthy to mention that testing, as we have done it, does not prove the system correct, but may only indicate the absence of problems with the cases that have been tested.

3. Characteristics of the Interface Software

In order for the DL/I interface to be successful, we have realized that it has to be well designed and well structured. Hence, we are cognizant of certain characteristics that the interface has to possess. Specifically, it has to be simple. In other words, it has to be easy to read and comprehend. The C code we have written has this characteristic. For instance, we often write the code with extra lines to avoid shorthand notations available in C. These extra lines have made the difference between comprehensible code and cryptic notations.

The interface software also has to be understandable. This has to be true to the extent that a maintenance programmer, for example, may easily grasp the functionality of the interface and the relation between it, and the other portions of the system. Our software

possesses this characteristic and does not have any hidden side-effects that could pose problems months or years from now. As a matter of fact, we have intentionally minimized the interaction between procedures to alleviate this problem.

The interface also has to be maintainable. This is important in light of the fact that almost 70% of all software life-cycle costs are incurred after the software becomes operational, i.e., in the maintenance phase. There are software engineering techniques we employed that have given the DL/I interface this characteristic. For example, we require programmers to update documentation of the interface code when changes are made. Hence, maintenance programmers have current documentation at all times. The problem of trying to identify the functionality of a program with dated documentation is alleviated. We also require the programmers to update their SSL specification as the code is being changed. Thus, the SSL specification consistently corresponds to the actual code. In addition, the data structures are designed to be general. Thus, it is an easy task to modify or rectify these structures to meet the demands of an evolving system.

The research conducted by Demurjian and Hsiao [Ref. 1] provides a high-level specification of the MLDS. The thesis written by Weishar [Ref. 3] extends the above work and provides a more detailed specification of a DL/I

language interface. This thesis outlines the actual implementation of a DL/I interface. The appendices provide the SSL design for this implementation.

A final characteristic that a DL/I interface should have is extensibility. A software product has to be designed in a manner that permits the easy modification and addition of code. In this light, we have placed "stubs" in appropriate locations within the KFS to permit easy insertion of the code needed to handle multiple horizontal screens of output. In addition, we have designed our data structures in such a manner that subsequent programmers may easily extend them, to handle not only multiple users, but also other language interfaces.

C. A CRITIQUE OF THE DESIGN

Our implementation of the DL/I interface possesses all of the elements of a successful software product. As noted previously, it is simple, understandable, maintainable, and extensible. Our constant employment of modern software engineering techniques have ensured its success.

However, there are two techniques that are especially worthy of critique. The first of these is the use of the SSL. Initially, we have felt that the implementation language may also serve as the language to specify program algorithms. However, in doing so, we have stifled our creativity. This is because we are concentrating not only

on what the algorithm does, but also on what the constructs (data structures) of the algorithm are. The use of the SSL has permitted us to concentrate on the functionality of the algorithm without a heavy concentration on its particular constructs. This has allowed us to view the algorithm in a detached manner so that the most efficient implementation for the constructs may be used. Although we have initially felt that the development of the program with the SSL may be too time-consuming, our opinions have changed when we have realized the advantages of the SSL and the overall complexity of the DL/I language interface.

The way in which the data structures are designed is the other noteworthy software engineering technique. Being relatively inexperienced programmers, we are inclined to use static structures. Hence, we have made extensive use of structures which are bound at compile time. We soon realize that in doing so, the computing resources of the system (i.e., the data space) are being depleted quite rapidly. Therefore, it is necessary for us to design the data structures in such a way that they may be managed in a dynamic fashion. Most of the data structures of the DL/I interface are linked-lists. This design affords us the most convenient way to efficiently utilize the resources of the system. It is an easy task to use the C language's malloc() (memory allocate) function to dynamically create the elements of a list as we need them. In addition, the free()

command is useful in releasing these same elements to be used again.

D. THE DATA STRUCTURES

The DL/I language interface has been developed as a single-user system that at some point is to be updated to a multi-user system. Two different concepts of the data are used in the language interface: (1) data shared by all users, and (2) data specific to each user. The reader should realize that the data structures used in our interface, and described below, have been deliberately made generic. Hence, these same structures support not only our DL/I interface, but the other language interfaces as well, i.e., SQL, CODASYL-DML, and Daplex.

1. Data Shared by All Users

The data structures that are shared by all users, are the database schemas defined by the users thus far. In our case, these are hierarchical schemas, consisting of segments and attributes. These are not only shared by all users, but also shared by the four modules of the MLDS, i.e., LIL, KMS, KC, and KFS. Figure 4 depicts the first data structure used to maintain data. It is important to note that this structure is represented as a union. Hence, it is generic in the sense that a user may utilize this structure to support SQL, DL/I, CODASYL-DML, or Daplex needs. However, we concentrate only on the hierarchical

```
union  dbid_node
  {
    struct  rel_dbid_node  *rel;
    struct  hie_dbid_node  *hie;
    struct  net_dbid_node  *net;
    struct  ent_dbid_node  *ent;
  }
```

Figure 4. The dbid_node Data Structure.

model. In this regard, the second field of this structure points to a record that contains information about a hierarchical database. Figure 5 illustrates this record. The first field is simply a character array containing the name of the hierarchical database. The next field contains an integer value representing the number of segments in the database. The third and fourth fields are pointers to hrec_node records, containing information about each segment in the database. Specifically, the third field points to the root segment in the database, while the fourth field points to the current segment being accessed. The final field is simply a pointer to the next database.

```
struct  hie_dbid_node
  {
    char                  name[DBNLength + 1];
    int                   num_seg;
    struct  hrec_node     *root_seg;
    struct  hrec_node     *curr_seg;
    struct  hie_dbid_node *next_db;
  }
```

Figure 5. The hie_dbid_node Data Structure.

The hrec_node record is shown in Figure 6, and contains information about each segment in the database. The first field of the record holds the name of the segment. The next field contains the number of attributes in the segment. The third and fourth fields point to hattr_node records which contain data on the first and current attributes of the segment. The next three fields point to other records of the same type. They give the schema its hierarchical form, pointing to a given segment's parent, first child, and next sibling. And finally, the last two fields contain the number of children and siblings that exist for the given segment in the hierarchical database schema.

Figure 7 shows the hattr_node record; the final record type used to support the definition of the hierarchical database schema. The first field is an array

```
struct  hrec_node
  {
    char                name[RNLength + 1];
    int                 num_attr;
    struct  hattr_node  *first_attr;
    struct  hattr_node  *curr_attr;
    struct  hrec_node   *parent;
    struct  hrec_node   *first_child;
    struct  hrec_node   *next_sib;
    int                 num_child;
    int                 num_sib;
  }
```

Figure 6. The hrec_node Data Structure.

```
struct  hattr_node
  {
    char                name[ANLength + 1];
    char                type;
    int                 length;
    int                 key_flag;
    int                 multiple;
    struct  hattr_node  *next_attr;
  }
```

Figure 7. The hattr_node Data Structure.

which holds the name of the attribute. The second field serves as a flag to indicate the attribute type. For instance, an attribute may either be an integer, a floating point number, or a string. The characters "i", "f", and "s" are used, respectively. The third field indicates the maximum length that a value of this attribute type may possibly have. For example, if this field is set to ten and the type of this attribute is a string, then the maximum number of characters that a value of this attribute type may have is ten. The fourth field is a flag used to indicate whether this particular attribute is the sequence field of the segment. The fifth field is a flag used to indicate whether twin segment occurrences of this type may contain the same sequence field values. The last field simply points to the next attribute in this segment. The reader may refer to Appendices B through E to examine the use of these data structures in the SSL.

2. Data Specific to Each User

This category of data represents information required to support each user's particular interface needs. The data structures used to accomplish this may be thought of as forming a hierarchy. At the root of this hierarchy is the user_info record, shown in Figure 8, which maintains information on all current users of a particular language interface. The user_info record holds the ID of the user, a union that describes a particular interface, and a pointer to the next user. The union field is of particular interest to us. As noted earlier, a union serves as a generic data structure. In this case, the union may hold the data for a user accessing either an SQL language interface layer, a DL/I LIL, a CODASYL-DML LIL, or a Daplex LIL. The li_info union is shown in Figure 9.

We are only interested in the data structures containing user information that pertain to the DL/I, or hierarchical, language interface. This structure is referred to as dli_info and is depicted in Figure 10. The

```
struct  user_info
  {
    char                uid[UIDLength + 1];
    union   li_info     li_type;
    struct  user_info   *next_user;
  }
```

Figure 8. The user_info Data Structure.

```
union  li_info
  {
    struct  sql_info  li_sql;
    struct  dli_info  li_dli;
    struct  dml_info  li_dml;
    struct  dap_info  li_dap;
  }
```

Figure 9. The li_info Data Structure.

first field of this structure, curr_db, is itself a record and contains currency information on the database being accessed by a user. The second field, file, is also a record. The file record contains the file descriptor and file identifier of a file of DL/I transactions, i.e., either requests or database descriptions. The next field, dli_tran, is also a record, and holds information that

```
struct  dli_info
  {
    struct  curr_db_info      curr_db;
    struct  file_info         file;
    struct  tran_info         dli_tran;
    int                       operation;
    struct  ddl_info          *ddl_files;
    union   kms_info          kms_data;
    union   kfs_info          kfs_data;
    int                       error;
    int                       answer;
    struct  hrec_node         saved_seg_ptr;
    struct  hrec_node         saved_seg_ptr2;
    struct  Sit_info          *kms_sit;
    struct  Sit_info          *sit_list;
    struct  Sit_status_info   *fst_sit_pos;
    struct  Sit_status_info   *curr_sit_pos;
    int                       buff_count;
  }
```

Figure 10. The dli_info Data Structure.

describes the DL/I transactions to be processed. This includes the number of requests to be processed, the first request to be processed, and the current request being processed. The fourth field of the dli_info record, operation, is a flag that indicates the operation to be performed. This may be either the loading of a new database or the execution of a request against an existing database. The next field, ddl_files, is a pointer to a record describing the descriptor and template files. These files contain information about the ABDL schema corresponding to the current hierarchical database being processed, i.e., the ABDL schema information for a newly defined hierarchical database. The following fields, kms_data and kfs_data, are unions that contain information required by the KMS and KFS. These are described in more detail in the next four chapters. The next field, error, is an integer value representing a specific error type. The next field, answer, is used by the LIL to record answers received through its interaction with the user of the interface. The next two fields, saved_seg_ptr and saved_seg_ptr2, are used by the KMS to save a pointer to the segment in the hierarchical schema that is last referenced during a previous call to the KMS. The first field is used by the KMS parser/translator, and the second field is utilized during the semantic analysis in the KMS. The next two fields, kms_sit and sit_list, are pointers to records that implement the status

information table (SIT), as discussed by Weishar [Ref. 3: pp. 32-36]. They allow the current position of the database to be maintained. They contain the ABDL equivalents of the DL/I requests, as well as result files to hold data retrieved from MBDS by these ABDL requests. The next two fields, fst_sit_pos and curr_sit_pos, contain information required by the KC to guide it in the execution of the translated DL/I requests. The last field, buff_count, is a counter variable used in the KC to keep track of the result buffers.

E. THE ORGANIZATION OF THE NEXT FOUR CHAPTERS

The following four chapters are meant to provide the user with a more detailed analysis of the modules constituting the MLDS. Each chapter begins with an overview of what each particular module does and how it relates to the other modules. The actual processes performed by each module are then discussed. This includes a description of the actual data structures used by the modules. Each chapter concludes with a discussion of module shortcomings.

III. THE LANGUAGE INTERFACE LAYER (LIL)

The LIL is the first module in the DL/I mapping process, and is used to control the order in which the other modules are called. The LIL allows the user to input transactions from either a file or the terminal. A transaction may take the form of either a database description (DBD) of a new database, or a DL/I request against an existing database. A transaction may contain multiple requests. This allows a group of requests that perform a single task, such as a looping construct in DL/I, to be executed together as a single transaction. The mapping process takes place when the LIL sends a single transaction to the KMS. After the transaction has been received by the KMS, the KC is called to process the transaction. Control always returns to the LIL, where the user may close the session by exiting to the operating system.

The LIL is menu-driven. When the transactions are read from either a file or the terminal, they are stored in a data structure called hie_req_info. If the transactions are DBDs, they are sent to the KMS in sequential order. If the transactions are DL/I requests, the user is prompted by another menu to selectively choose an individual request to be processed. The menus provide an easy and efficient way

for the user to view and select the methods of request processing desired. Each menu is tied to its predecessor, so that by exiting one menu the user is moved up the "menu tree". This allows the user to perform multiple tasks in one session.

A. THE LIL DATA STRUCTURES

The LIL uses two data structures to store the user's transactions and control which transaction is to be sent to the KMS. It is important to note that these data structures are shared by both the LIL and the KMS.

The first data structure is named tran_info and is shown in Figure 11. The first field of this record, first_req, contains the address of the first transaction that has been read from a file or the terminal. The second field, curr_req, contains the address of the transaction currently being processed. The LIL sets this pointer to the transaction that the KMS is to process next, and then calls the KMS. The third field, no_req, contains the number of

```
struct  tran_info
    {
      struct  hie_req_info  *first_req;
      struct  hie_req_info  *curr_req;
      int                   no_req;
    }
```

Figure 11. The tran_info Data Structure.

transactions currently in the transaction list. This number is used for loop control when printing the transaction list to the screen, or when searching the list for a transaction to be executed.

The second data structure used by the LIL is named hie_req_info. Each copy of this record represents a user transaction, and thus, is an element of the transaction list. The hie_req_info record is shown in Figure 12. The first field of this record, req, is a character string that contains the actual DL/I transaction. The second field, in_req, is a pointer to a list of character arrays that each contain a single line of one transaction. After all lines of a transaction have been read, the line list is concatenated to form the actual transaction, req. The third field of this record, req_len, contains the length of the transaction. It is used to allocate the correct and minimal amount of memory space for the transaction. If a transaction contains multiple requests, the fourth field,

```
struct  hie_req_info
   {
     char                    *req;
     struct  temp_str_info   *in_req;
     int                     req_len;
     struct  hie_req_info    *sub_req;
     struct  hie_req_info    *next_req;
   }
```

Figure 12. The hie_req_info Data Structure.

sub_req, points to the list of requests that make up the transaction. In this case, the field in_req is the first request of the transaction. The last field, next_req, is a pointer to the next transaction in the list of transactions.

B. FUNCTIONS AND PROCEDURES

The LIL makes use of a number of functions and procedures in order to create the transaction list, pass elements of the list to the KMS, and maintain the database schemas. We do not describe each of these functions and procedures in detail. Rather, we provide a general description of the LIL processes.

1. Initialization

The MLDS is designed to be able to accommodate multiple users, but is implemented to support only a single user. To facilitate the transition from a single-user system to a multiple-user system, each user possesses his own copy of a user data structure when entering the system. This user data structure stores all of the relevant data that the user may need during their session. All four modules of the language interface make use of this structure. The modules use many temporary storage variables, both to perform their individual tasks, and to maintain common data between modules. The transactions, in user data language form, and mapped kernel data language form, are also stored in each user data structure. It is

easy to see that the user structure provides consolidated, centralized control for each user of the system. When a user logs onto the system, a user data structure is allocated and initialized. The user ID becomes the distinguishing feature to locate and identify different users. The user data structures for all users are stored in a linked-list. When new users enter the system, their user data structures are appended to the end of the list. In our current environment there is only a single element on the user list. In a future environment, when there are multiple users, we simply expand the user list as described above.

2. Creating the Transaction List

There are two operations the user may perform. A user may define a new database or process DL/I requests against an existing database. The first menu that is displayed prompts the user to select the operation desired. Each operation represents a separate procedure to handle specific circumstances. The menu looks like the following:

```
Enter type of operation desired
     (l) - load a new database
     (p) - process old database
     (x) - return to the operating system
ACTION ----> _
```

For either choice (i.e., l or p), another menu is displayed to the user requesting the mode of input. This input may always come from a data file. If the operation

selected from the previous menu had been "p", then the user may also input transactions interactively from the terminal. The generic menu looks like the following:

```
Enter mode of input desired
     (f) - read in a group of transactions from a file
     (t) - read in transactions from the terminal
     (x) - return to the previous menu
ACTION ----> _
```

Note that the "t" choice would be omitted if the operation selected from the previous menu had been to load a new database. Again, each mode of input selected corresponds to a different procedure to be performed. The transaction list is created by reading from the file or terminal, looking for an end-of-transaction marker or an end-of-file marker. These flags tell the system when one transaction has ended, and when the next transaction begins. When the list is being created, the pointers to access the list are initialized. These pointers, first_req and curr_req, have been described earlier in the data structure subsection. Both pointers are set to the first transaction read, in other words, the head of the transaction list.

3. Accessing the Transaction List

Since the transaction list stores both DBDs and DL/I requests, two different access methods have to be employed to send the two types of transactions to the KMS. We discuss the two methods separately. In both cases, the KMS

accesses a single transaction from the transaction list. It does this by reading the transaction pointed to by the request pointer, curr_req, of the tran_info data structure (see Figure 11). Therefore, it is the job of the LIL to set this pointer to the appropriate transaction before calling the KMS.

a. Sending DBDs to the KMS

When the user specifies the filename of DBDs (input from a file only) further user intervention is not required. To produce a new database, the transaction list of DBDs is sent to the KMS via a program loop. This loop traverses the transaction list, calling the KMS for each DBD in the list.

b. Sending DL/I Requests to the KMS

In this case, after the user has specified the mode of input, the user conducts an interactive session with the system. First, all DL/I requests are listed to the screen. As the requests are listed from the transaction list, a number is assigned to each transaction in ascending order, starting with the number one. The number appears on the screen to the left of the first line of each transaction. Note that each transaction may contain multiple requests. Next, an access menu is displayed which looks like the following:

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding transactions
     (d)   - redisplay the list of transactions
     (r)   - reset the currency pointer to the root
     (x)   - return to the previous menu
ACTION ----> _
```

Since DL/I requests are independent items, the order in which they are processed does not matter. The user has the option of executing any number of DL/I requests. A loop causes the menu to be redisplayed after any DL/I request has been executed so that further choices may be made. The "r" selection causes the currency pointer to be repositioned to the root of the hierarchical schema so that subsequent requests may access the complete database, rather than be limited to beginning from a current position established by previous requests.

4. Calling the KC

As mentioned earlier, the LIL acts as the control module for the entire system. When the KMS has completed its mapping process, the transformed transactions have to be sent to the KC to interface with the kernel database system. For DBDs, the KC is called after all DBDs on the transaction list have been sent to the KMS. The mapped DBDs have been placed in a mapped transaction list that the KC is going to access. Since DL/I requests are independent items, the user should wait for the results from one DL/I request before issuing another. Therefore, after each DL/I request

has been sent to the KMS, the KC is immediately called. The mapped DL/I requests are placed on a mapped transaction list, which the KC may easily access.

5. Wrapping-up

Before exiting the system, the user data structure described in Chapter II has to be deallocated. The memory occupied by the user data structure is freed and returned to the operating system. Since all of the user structures reside in a list, the exiting user's node has to be removed from the list.

IV. THE KERNEL MAPPING SYSTEM (KMS)

The KMS is the second module in the DL/I mapping interface and is called from the language interface layer (LIL) when the LIL has received DL/I requests input by the user. The function of the KMS is to: (1) parse the request to validate the user's DL/I syntax, (2) translate, or map, the request to equivalent ABDL request(s), and (3) perform a semantic analysis of the current ABDL request(s) generated, relative to the request(s) generated during a previous call to the KMS. Once an appropriate ABDL request, or set of requests, has been formed, it is made available to the kernel controller (KC) which then prepares the request for execution by MBDS. The KC is discussed in Chapter V.

A. AN OVERVIEW OF THE MAPPING PROCESS

From the description of the KMS functions above we immediately see the requirement for a parser as a part of the KMS. This parser validates the DL/I syntax of the input request. The parser grammar is the driving force behind the entire mapping system.

1. The KMS Parser / Translator

The KMS parser has been constructed by utilizing Yet-Another-Compiler Compiler (YACC) [Ref. 15]. YACC is a program generator designed for syntactic processing of token

input streams. Given a specification of the input language structure (a set of grammar rules), the user's code to be invoked when such structures are recognized, and a low-level input routine, YACC generates a program that syntactically recognizes the input language and allows invocation of the user's code throughout this recognition process. The class of specifications accepted is a very general one: LALR(1) grammars. It is important to note that the user's code mentioned above is our mapping code that is going to perform the DL/I-to-ABDL translation. As the low-level input routine, we utilize a Lexical Analyzer Generator (LEX) [Ref. 16]. LEX is a program generator designed for lexical processing of character input streams. Given a regular-expression description of the input strings, LEX generates a program that partitions the input stream into tokens and communicates these tokens to the parser.

The parser produced by YACC consists of a finite-state automaton with a stack and performs a top-down parse, with left-to-right scan and one token look-ahead. Control of the parser begins initially with the highest-level grammar rule. Control descends through the grammar hierarchy, calling lower and lower-level grammar rules which search for appropriate tokens in the input. As the appropriate tokens are recognized, some portions of the mapping code may be invoked directly. In other cases, these tokens are propagated back up the grammar hierarchy until a

higher-level rule has been satisfied, at which time further translation is accomplished. When all of the necessary lower-level grammar rules have been satisfied and control has ascended to the highest-level rule, the parsing and translation processes are complete. In Section B, we give an illustrative example of these processes. We also describe the subsequent semantic analysis necessary to complete the mapping process.

2. The KMS Data Structures

The KMS utilizes, for the most part, just five structures defined in the interface. It, naturally, requires access to the DL/I input request structure discussed in Chapter II, the dli_tran structure. However, the five data structures to be discussed here are only those unique to the KMS.

The first of these, shown in Figure 13, is a record that contains information accumulated by the KMS during the grammar-driven parse that is not of immediate use. This record allows the information to be saved until a point in the parsing process where it may be utilized in the appropriate portion of the translation process. The first field in this record, tgt_list, is a pointer to the head of a list of attribute names. These are the names of those attributes whose values are retrieved from the database. The second field, insert_list, is a pointer to the head of a list of insert lists. This list generally contains a single

```
struct  hie_kms_info
  {
    struct  symbolic_id_info  *tgt_list;
    struct  insert_lists      *insert_list;
    char                      *temp_str;
  }
```

Figure 13. The hie_kms_info Data Structure.

item, which points to a single insert list. However, in the case of multiple path insertion (i.e., ISRT, specifying command code D), this list contains an item that points to each insert list, corresponding to each segment to be inserted. Each insert list, then, holds the values that an ISRT request desires inserted into the database for a given segment. The third field, temp_str, is a pointer to a variable-length character string. The character-string length is a function of the input request length, and is allocated, when required, to accumulate intermediate translation results while parsing the boolean predicates that optionally follow the segment name in the segment search argument (SSA) of a given user request.

The next three data structures, shown in Figure 14, are records that are pointed to by the hie_kms_info record, as just described. Respectively, they represent a list of attribute names (the target list), a list of insert list nodes, and a list of attribute values (the insert list(s)). ANLength and RNLength are constants defining the maximum lengths of attribute and segment names, respectively. Each

```
struct  symbolic_id_info
  {
    char                        name[ANLength + 1];
    int                         length;
    struct  symbolic_id_info    *next_attr;
  }

struct  insert_lists
  {
    char                    *list;
    int                     insert_attrs;
    int                     insert_vals;
    char                    seg_name[RNLength + 1];
    struct  hrec_node       *seg_ptr;
    struct  insert_lists    *next_list;
  }

struct  insert_info
  {
    char                  attr[ANLength + 1];
    char                  *value;
    char                  type;
    struct  insert_info   *next_val;
  }
```

Figure 14. Additional KMS Data Structures.

insert_lists node contains the number of insert_attrs (attributes to be inserted) and the number of insert_vals (values to be inserted) for a given insert list, as well as the segment name and a pointer to that segment in the hierarchical schema. Each insert_info item contains the attribute name, attribute value, and type information corresponding to the item that is to be inserted into the database. It should be noted that the value field in the insert_info record is a pointer to a variable-length character string. Although attribute names have a constant maximum-length constraint, the length of attribute values in

the database is limited only by the constraint placed on them by the user in the original database definition, and as such, they may be of varying lengths.

The remaining KMS data structure, shown in Figure 15, is our implementation of the status information table (SIT) discussed by Weishar [Ref. 3: pp. 32-36]. The KMS, in general, maps a single DL/I request to multiple ABDL requests. We require one Sit_info record corresponding to each of these ABDL requests. The first two fields, prev and next, are pointers to other records of the same type that connect the records in a linearly-linked list. The next three fields, parent, child and sibling, are pointers that interconnect the records in a hierarchical manner. These

```
struct  Sit_info
  {
    struct  Sit_info        *prev;
    struct  Sit_info        *next;
    struct  Sit_info        *parent;
    struct  Sit_info        *child;
    struct  Sit_info        *sibling;
    struct  Sit_info        *loop;
    struct  Sit_info        *nf_loop;
    char                    *abdl_req;
    int                     operation;
    int                     cmd_code;
    int                     or;
    char                    *template;
    char                    seg_name[RNLength + 1];
    int                     BOR;
    int                     EOR;
    struct  hie_file_info   *result_file;
  }
```

Figure 15. The Sit_info Data Structure.

pointers are required because some multiple ABDL requests are generated into a linear list by the KMS, through a depth-first search of the hierarchical schema, i.e., a tree walk that effectively flattens the tree. However, such multiple requests have to be processed by the KC in a hierarchical, rather than linear, fashion. Thus, the parent, child and sibling pointers preserve the hierarchical form of the linear list, i.e., the flattened tree. The following two fields, loop and nf_loop, are pointers that indicate a looping construct in the DL/I input request, i.e., a "label name" declaration, accompanied by a GOTO "label name" statement. The next field, abdl_req, is a pointer to the actual ABDL request generated by the KMS. The following two fields, operation and cmd_code, are flags indicating the DL/I operation desired (e.g., GU, GN, GNP, ..., etc.), and which command code, if any, is resident in the DL/I source request. The next field, or, indicates whether there is an "or" in the resulting ABDL request. This is used by the KC during the completion of ABDL requests that may not be fully-formed by the KMS. The KC uses the next field, template, as working space for these purposes. The following field, seg_name, contains the segment name of the translated request. The next two fields, BOR and EOR, mark the beginning and end of multiple ABDL requests for control purposes in the KC. The last

field, result_file, is used in the KC to accumulate results obtained from MBDS when executing the ABDL requests.

At the end of the mapping process, before control is surrendered to the LIL, all data structures that are unique to the KMS which have been allocated during the mapping process are returned to the free list.

B. FACILITIES PROVIDED BY THE IMPLEMENTATION

In this section, we discuss those DL/I facilities that are provided by our implementation of the hierarchical interface. We do not discuss the DL/I-to-ABDL translation in detail. Rather, we provide an overview of the salient features of the KMS, accompanied by one illustrative example of the parsing and translation processes. User-issued requests may take two forms, DL/I database definitions, or DL/I database manipulations. In the case of database manipulations, we also describe the semantic analysis necessary to complete the mapping process. Appendix C contains the design of our implementation, written in a system specification language (SSL).

1. Database Definitions

When the user informs the LIL that the user wishes to create a new database, the job of the KMS is to build a hierarchical database schema that corresponds to the database definition input by the user. The LIL initially allocates a new database identification node (hie_dbid_node

shown in Figure 5) with the name of the new database, as input by the user. The LIL then sends the KMS a complete database definition, which takes the form of a DL/I database description (DBD) as follows:

```
DBD   NAME= database_name
SEGM  NAME= segment_1
FIELD NAME= (attr_1,SEQ[,M]), [TYPE=type,] BYTES= length
FIELD NAME=  attr_2, [TYPE=type,] BYTES= length
               .
               :
FIELD NAME=  attr_i, [TYPE=type,] BYTES= length
SEGM  NAME= segment_2
FIELD NAME= (attr_1,SEQ[,M]), [TYPE=type,] BYTES= length
FIELD NAME=  attr_2, [TYPE=type,] BYTES= length
               .
               :
FIELD NAME=  attr_j, [TYPE=type,] BYTES= length
SEGM  NAME= segment_3
  .
  :
SEGM  NAME= segment_n
```

The sequence of statements in the DBD is significant. Specifically, SEGM statements have to appear in the sequence that reflects the hierarchical structure, i.e., top to bottom, left to right. Also, each SEGM statement has to be immediately followed by the appropriate FIELD statements. The FIELD statement for the sequence field (indicated in the DBD example by SEQ) has to be the first such statement for the segment. The sequence field is taken to be unique, unless M (multiple) is specified. If M is specified, two occurrences of the given segment type may have the same value for the sequence field. If the optional TYPE specification is omitted, the data type CHAR is the

default. For each SEGM statement, an additional segment node (hrec_node shown in Figure 6) is added to the database schema under construction. For each subsequent FIELD statement, an additional attribute node (hattr_node shown in Figure 7) is added to the schema for the current segment under construction. The database identification node (hie_dbid_node shown in Figure 5) holds the number of segments in the schema and the database name, each segment node holds the number of attributes in that segment and the segment name, and each attribute node holds the attribute name, type, length, and sequence field information.

When the KMS has parsed all the statements included in the DBD, the result is a completed database schema, as shown in Figure 16. Not shown in Figure 16, is the list of attribute nodes that is connected to each segment node. The hierarchical database schema, when completed, serves two purposes. First, when creating a new database, it facilitates the construction of the MBDS template and descriptor files. Secondly, when processing requests against an existing database, it allows a validity check of the segment and attribute names. It also serves as a source of information for type-checking.

2. Database Manipulations

When the user wishes the LIL to process requests against an existing database, the first task of the KMS is to map the user's DL/I request to equivalent ABDL requests.

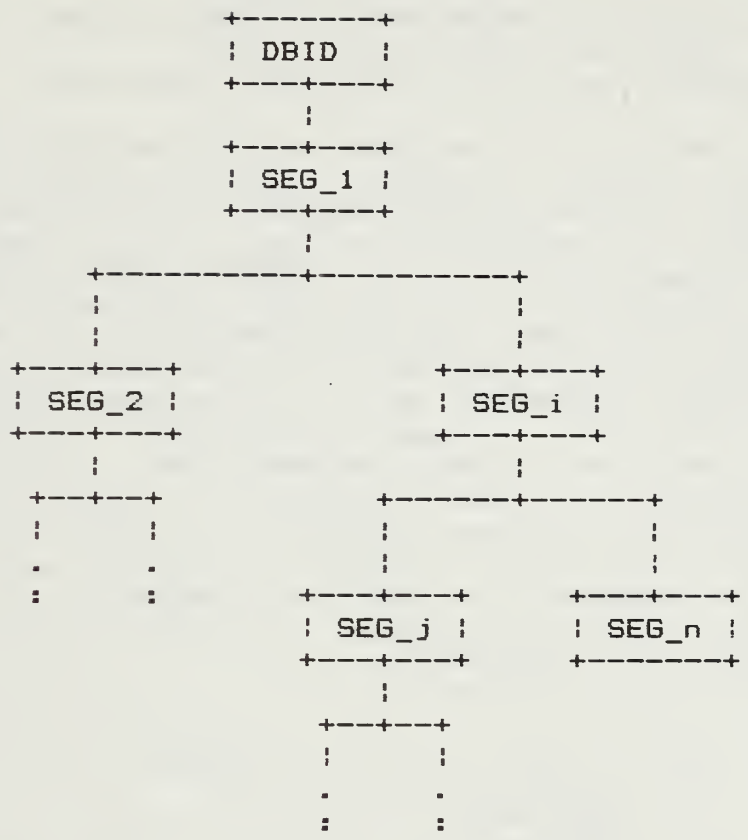


Figure 16. The Hierarchical Database Schema.

a. The DL/I GET Calls to the ABDL RETRIEVE

The DL/I GET calls consist of the Get Unique (GU), Get Next (GN), and Get Next within Parent (GNP) operations. The fact that each of these calls is quite different in functionality is of little concern to the KMS parser/translator. All of these calls have identical form, syntactically, with the exception of the DL/I operator, i.e., GU, GN, GNP. Therefore, the KMS maps each DL/I GET call to an equivalent ABDL RETRIEVE request, or, as in most cases, a series of ABDL RETRIEVE requests. An operator

identification flag is set during the translation process which allows the KC to associate the appropriate operation to these requests for controlling their execution.

The DL/I GU operation is a direct retrieval, and as such, has to specify the complete hierarchical path to the desired segment. That is, it specifies the segment type at each level of the database, from the root down to the desired segment, together with an optional occurrence-identifying condition for each segment type. (Collectively, such a specification, at each level, is referred to as a segment search argument for that level/segment.) An example of such a call is as follows:

```
GU  course (ctitle = 'mlds')
    offering
    student
```

This call retrieves information concerning the first occurrence of a student enrolled in the course entitled "mlds". The series of ABDL requests generated for such a call is as follows:

```
[ RETRIEVE  ((TEMPLATE = COURSE) and
             (CTITLE = Mlds))
            (CNUM) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****))
            (DATE) BY DATE ]
```

```
[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (CNUM = ****) and
             (DATE = ******))
            (SUM, SNAME, GRADE) BY SNUM ]
```

Notice that only the first RETRIEVE request generated is fully-formed, i.e., may be submitted to MBDS "as-is". Subsequent requests may not be completed until the appropriate sequence-field values have been obtained from the execution of previous requests. This process takes place in the KC. The KMS uses asterisks, as place holders, to mark the maximum allowable length of such sequence fields. Each RETRIEVE request, with the exception of the last, is generated solely to extract the hierarchical path to the desired segment. (By so doing, they allow the KC to establish and maintain the current position within each segment referenced in a DL/I call.) Consequently, the only attribute in each target list is that of the segment sequence field. Of course, the target list of the last request contains all the attributes of the desired segment. It is the information obtained from the execution of the final request which is returned to the user, via the KFS. Also of note is that each request includes the optional ABDL "BY attribute_name" clause. The work of Weishar [Ref. 3: pp. 39-42] has proposed that the results obtained from each RETRIEVE request would be sorted by sequence-field value in the language interface. We chose to let the KDS

(i.e., MBDS) perform this operation through the inclusion of a "BY sequence_field" clause on all ABDL RETRIEVE requests.

The DL/I GN and GNP operations are sequential retrievals. As such, they may each contain a looping construct. Such a construct takes the form of a label that precedes the GN or GNP operator, and a GOTO statement following the last segment search argument of the DL/I call. GN and GNP operations are predicated on the fact that a previous DL/I call has established a current position within the database. Therefore, unlike the GU operation, they need not specify the complete hierarchical path from the root to the desired segment. This does, however, make it necessary to semantically analyze the GN or GNP, and the previous DL/I call. This analysis is discussed in Subsection 3. An example of such a call is as follows:

```
xx   GNP  student
     GOTO  xx
```

This call retrieves information concerning the next occurrence of a student enrolled in the course, and the offering of that course, which have been established as the current COURSE and OFFERING segments within the database by the previous GET operation (of any type) or ISRT operation. The ABDL request generated for such a call is as follows:

```
[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (CNUM = ****) and
             (DATE = ******))
            (SNUM, SNAME, GRADE) BY SNUM ]
```

There is no indication, from the ABDL request generated, that the DL/I call contained a looping construct. However, a loop pointer is set during the translation process which allows the KC to discern that a looping construct exists and the extent of such a construct. The KMS translator recognizes that the first segment search argument of this DL/I call does not specify the root segment as its segment type. Consequently, it performs a tree walk of the hierarchical schema, in reverse order, to obtain the sequence fields required to complete the translation process, i.e., those that specify the complete path from the root to the segment concerned; in this case, CNUM and DATE.

The GN and GNP operators may also be used to perform sequential retrieval without the specification of SSAs. In the case of the GN operator, such a call retrieves all of the segments (of all types) subordinate to the last segment type referenced in the previous DL/I call, which established the current position within the database. The GNP operator functions similarly, except that instead of retrieving all subordinate segments, it only retrieves subordinate child segments, i.e., it does not retrieve segments below the immediate children of the current parent

segment. Since no SSA is specified, the KMS translator has to save the identity of the last segment type referenced in each DL/I call. Since the KMS does not know when it might receive such a DL/I call, this allows the translator to identify where the sequential retrieval begins for such a DL/I call, i.e., which segment types constitute "subordinate" segments. An example of such a call is as follows:

```
yy   GN
     GOTO  yy
```

Assuming that the previous DL/I call is simply "GU course", the series of ABDL requests generated are as follows:

```
[ RETRIEVE  ((TEMPLATE = PREREQ) and
             (CNUM = ****))
            (PNUM, PTITLE) BY PNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****))
            (DATE, LOCATION, FORMAT) BY DATE ]

[ RETRIEVE  ((TEMPLATE = TEACHER) and
             (CNUM = ****) and
             (DATE = ******))
            (TNUM, TNAME) BY TNUM ]

[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (CNUM = ****) and
             (DATE = ******))
            (SNUM, SNAME, GRADE) BY SNUM ]
```

If subordinate segments for PREREQ, TEACHER or STUDENT were in the database, appropriate RETRIEVE requests would have

been generated for those segment types also, i.e., PREREQ, TEACHER and STUDENT are the leaves of our example database. However, if our example DL/I call contained GNP, instead of GN, only the RETRIEVEs for PREREQ and OFFERING would be generated, i.e., only the children of COURSE in our example database. Also, notice that each request includes all attributes for that segment type in its target list. That is, complete information about each of these segments is to be returned to the user.

b. The DL/I GET HOLD Calls to the ABDL RETRIEVE

The DL/I GET HOLD calls consist of the Get Hold Unique (GHU), Get Hold Next (GHN), and Get Hold Next within Parent (GHNP) operations. A DL/I GET HOLD call is used to retrieve a given segment occurrence into a work area, and hold it there so that it may subsequently be updated or deleted. ABDL does not have this requirement. Therefore, when the KMS parser encounters one of these calls, the KMS translator treats them as a corresponding GET call. With the exception of the "H", the general form of the GET HOLD calls is identical to the forms of the non-HOLD (i.e., GET) counterparts. Thus, the mapping processes described in the previous subsection are applicable to the GET HOLD calls, with the exception of the special case of sequential retrieval without the specification of SSAs. Such a call has no meaning with a GET HOLD operator.

c. The DL/I DLET to the ABDL DELETE

The DL/I DLET consists of a GET HOLD call, together with the reserved word DLET immediately following the last SSA in the GET HOLD portion of the call. When the KMS parser encounters the GET HOLD portion of the call, the KMS translator generates the appropriate ABDL RETRIEVE requests. Then, when the reserved word DLET is parsed, the KMS translator generates appropriate ABDL DELETE requests to delete the current segment occurrence (i.e., for the current position just established by the GET HOLD portion of the call), as well as all of the children, grandchildren, etc. (i.e., the descendants) of the current segment occurrence. An example of such a call is as follows:

```
GHU  course (ctitle = 'mlds')
     offering
DLET
```

Assuming that there is only one offering of the course entitled "mlds", this call deletes the occurrences of that course and offering, along with all the teachers and students associated with them. The series of ABDL requests generated for such a call is as follows:

```
[ RETRIEVE  ((TEMPLATE = COURSE) and
             (CTITLE = Mlds))
            (CNUM) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****))
            (DATE) BY DATE ]

[ DELETE ((TEMPLATE = OFFERING) and
          (CNUM = ****) and
          (DATE = ******)) ]

[ DELETE ((TEMPLATE = TEACHER) and
          (CNUM = ****) and
          (DATE = ******)) ]

[ DELETE ((TEMPLATE = STUDENT) and
          (CNUM = ****) and
          (DATE = ******)) ]
```

In general, a single RETRIEVE request is generated for each SSA in the GET HOLD portion of the DL/I DLET. Then, for each segment type subordinate to the segment type referenced in the last SSA: (1) if the segment type is a leaf, a single ABDL DELETE is generated for that segment, and (2) if the segment type is not a leaf, a pair of ABDL requests are generated for that segment, one RETRIEVE and one DELETE. In our example, TEACHER and STUDENT are leaf segment types, and thus, no additional RETRIEVE requests are generated for those segment types. Notice that each RETRIEVE request simply retrieves the sequence-field attribute for the appropriate segment type. The sequence-field values are all that is required, since no information is returned to the user as a result of these RETRIEVE

requests. These are the values required to complete the DELETE requests, specifying the complete hierarchical path from the root to the segment to be deleted.

d. The DL/I REPL to the ABDL UPDATE

We are implementing DL/I in an interactive language interface. However, DL/I is an embedded database language that is invoked from a host language (i.e., PL/I, COBOL, or System/370 Assembler Language) by means of subroutine calls. The syntax for providing an appropriate attribute-value pair to be changed during a DL/I REPL call is resident in the host language, not in the DL/I data language itself. In order to make an embedded language function interactively, we are forced to introduce additional syntax for the language interface. This additional syntax does not represent a change to the DL/I data language, but rather, serves only to facilitate our interactive implementation of the normally embedded data language, DL/I. Therefore, we have implemented the following syntax in the DL/I REPL which allows the user to input the attribute-value pair they desire to change:

CHANGE attribute_name TO attribute_value

The DL/I REPL consists of a GET HOLD call, with our additional syntax immediately following the last SSA in the GET HOLD portion of the call, and then the reserved word REPL. When the KMS parser encounters the GET HOLD portion

of the call, the KMS translator generates the appropriate ABDL RETRIEVE requests. When the KMS parser encounters our additional syntax, it saves the attribute-value pair in local variables for subsequent use by the KMS translator. Then, when the reserved word REPL is parsed, the KMS translator generates the appropriate ABDL UPDATE request to update the current segment occurrence, i.e., for the current position just established by the GET HOLD portion of the call. An example of such a DL/I REPL call is as follows:

```
GHU  course (ctitle = 'mlds')
     prereq (ptitle = 'mdbs')
CHANGE  ptitle  TO  'mbds'
REPL
```

The series of ABDL requests generated for such a DL/I REPL call is as follows:

```
[ RETRIEVE  ((TEMPLATE = COURSE) and
             (CTITLE = Mlds))
            (CNUM) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = PREREQ) and
             (CNUM = ****) and
             (PTITLE = Mdbs))
            (PNUM) BY PNUM ]

[ UPDATE ((TEMPLATE = PREREQ) and
          (CNUM = ****) and
          (PNUM = ****))
         <PTITLE = Mbds> ]
```

Notice that each RETRIEVE request simply retrieves the sequence-field attributes for the appropriate segment type, i.e., like the DL/I DLET, no information is returned to the user as a result of these RETRIEVE requests. As is the case with ABDL, we may only update a single attribute value in each DL/I REPL call. However, each DL/I REPL call updates that particular attribute-value pair in all multiple record occurrences that may exist.

e. The DL/I ISRT to the ABDL INSERT

As in the case of the DL/I REPL, we are forced to introduce additional syntax to allow the DL/I ISRT to function in our interactive language interface. In this instance, we have implemented the following syntax for the DL/I ISRT, which allows the user to build the new segment to be inserted to the database:

BUILD [(attr_1, ..., attr_n)] : (value_1, ..., value_n)

If values are to be inserted for each attribute of the segment type, there is no requirement to list the attribute names. Only the attribute values need be listed. However, they have to appear in the same order in which they were defined during the original definition of the database. A value for the sequence-field attribute may not be omitted from the list. Due to the ABDL requirement that the INSERT request include values for all attributes, in the case where the user does not specify values for all attributes in the

segment, the KMS translator inserts default values. We use a zero (0) and a "Zz" as the default values for the data types integer and character, respectively.

The DL/I ISRT consists of our additional syntax to build a new segment occurrence, followed by a sequence of SSAs, the first of which is preceded by the reserved word ISRT. This sequence of SSAs has to specify the complete hierarchical path from the root to the segment to be inserted. An example of such a call is as follows:

```
build  (tnum, tname) : (1234, 'hsiao')
isrt   course (ctitle = 'mbds')
       offering (date = 850430)
       teacher
```

The series of ABDL requests generated for such a DL/I ISRT call is as follows:

```
[ RETRIEVE  ((TEMPLATE = COURSE) and
             (CTITLE = Mbds))
            (CNUM) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****) and
             (DATE = 850430))
            (DATE) BY DATE ]

[ INSERT (<TEMPLATE, TEACHER>,
          <CNUM, ****>,
          <DATE, ******>,
          <TNUM, 1234>,
          <TNAME, Hsiao>) ]
```

Although the sequence field of the OFFERING segment has been specified in its SSA, the translator does not recognize this fact. Therefore, the RETRIEVE request for the OFFERING segment is mechanically generated, in spite of the fact that we are given the value that we subsequently retrieve when executing this request. This RETRIEVE returns only one date, in this case, 850430. No RETRIEVE request is generated for the TEACHER segment. In general, no RETRIEVE request is generated for the last SSA in the DL/I ISRT. This is because the last SSA represents the segment to be inserted and, by definition, the user gives us the sequence-field value when building the new segment. The KMS translator did not have to insert any default values, as all TEACHER attributes have been listed by the user in building the new segment.

f. The Mapping Processes: An Example

In this subsection we present an illustrative example of the KMS mapping processes (i.e., parsing and translation) for a simple DL/I GU call. We begin by showing the grammar for the dml_statement portion of the KMS. We then step through the grammar and demonstrate appropriate portions of our design in system specification language (SSL). We only show those portions of the design that are relevant to the example, i.e., those that would actually be executed. The entire KMS design is shown in Appendix C.

The relevant grammar is shown in Figure 17. The source DL/I call to be utilized for our example is the following:

```
GU  course
```

The ABDL request generated in response to such a DL/I call is as follows:

```
[ RETRIEVE  (TEMPLATE = COURSE)
            (CNUM, CTITLE, DESCRIPN) BY CNUM ]
```

To begin our discussion, let us first synchronize the reader. At the beginning of the mapping process, the parse descends the grammar hierarchy searching for appropriate tokens in the input that may satisfy one of the grammar rules. Therefore, the parser descends through the ddl_statement rules (data definition language). After finding no matching tokens for those rules, the parser eventually descends to the dml_statement rule (data manipulation language).

First, when the dml_statement rule is called, it immediately calls the J rule. The J rule searches for a label in the input. Since no label exists in this DL/I call, the empty portion of the J rule is matched, satisfying the J rule. Control reverts to the dml_statement rule, which then immediately calls the ssa rule. The ssa rule

```
dml_statement:  J  ssa

J:  empty
 |  H

H:  IDENTIFIER
 |  VALUE

ssa:  seg_srch_arg
   |  ssa  seg_srch_arg

seg_srch_arg:
    dli_operator  segment_name  L  G  E  K
 |  dli_operator  E  K

segment_name:  IDENTIFIER

L:  empty
 |  ASTERISK  N

G:  empty
 |  LPAR  boolean  RPAR

E:  empty
 |  GOTO  H
 |  NFGOTO  H

K:  empty
 |  dli_op
```

Figure 17. The KMS dml_statement Grammar.

calls the seg_srch_arg rule, which then calls the dli_operator rule. The dli_operator rule (not shown in Figure 17) recognizes the GU token of the DL/I call, sets the operator_flag to signify a GU operation has been discovered, and returns control to the seg_srch_arg rule. The seg_srch_arg rule then calls the segment_name rule which recognizes the IDENTIFIER token (i.e., course) in the DL/I call and returns control to the seg_srch_arg rule.

Next, even though the seg_srch_arg rule is not completely satisfied, we need to perform some translation. The following SSL is invoked, before the L rule is called:

```
seg_srch_arg:
    dli_operator  segment_name
      {
      seg_ptr = the root segment of the db
      if (! valid_parent(seg_ptr, seg, curr_seg_ptr))
        print ("Error - segment_name does not exist")
        perform yyerror()
        return
      end_if
      if (operator_flag != ISRT)
        alloc and init the abdl_str and tgt_list item
        copy  "[ RETRIEVE  ("  to the abdl_str
        copy segment seq_fld and length to tgt_list
      end_if
        .
        :
      save segment_name for later use
      }
    L  G  E  K
  |  dli_operator  E  K
```

We set a pointer to the root of the database, which is then passed as an argument to the valid_parent() function. The valid_parent() function traverses the hierarchical schema to determine whether a segment type with the given segment_name exists, and returns true, along with a pointer to that segment type in the hierarchical schema (curr_seg_ptr), if found. Otherwise, valid_parent() returns false, in which case an error message is printed, an error routine is called, and then we simply return from the mapping process. Therefore, since COURSE is a valid segment name, we continue. The operator_flag has already been set to GU, so

we allocate and initialize the abdl_str to be used to accumulate the ABDL request, and the first tgt_list item to hold the sequence-field attribute to be retrieved. We then copy "[RETRIEVE (" to the abdl_str. We also access the schema, via the curr_seg_ptr set by valid_parent(), and obtain the segment sequence field and its maximum length, which is then stored in the tgt_list item just allocated. Finally, we save the value of the segment name in a local variable for later use.

We now continue with the seg_srch_arg rule, which calls the L rule. The L rule searches for a command code token in the input. Since no command code exists in this DL/I call, the empty portion of the L rule is matched, satisfying the L rule. Control reverts to the seg_srch_arg rule, which then calls the G rule. The G rule searches for a segment occurrence qualification (a boolean predicate) in the input. Since no such expression exists in this DL/I call, the empty portion of the G rule is matched, satisfying the G rule, and the following SSL is invoked:

```
G:  empty
      { .
        :
      if (curr_seg_ptr = the root of the db)
        concat "TEMPLATE = 'segment_name'" to abdl_str
      end_if
        .
        :
      }
  |  LPAR  boolean  RPAR
```

The curr_seg_ptr has previously been set to the COURSE segment, which is the root of our example database. Therefore, we concatenate "TEMPLATE = COURSE" to the abdl_str.

Now control returns to the seg_srch_arg rule, and the following SSL is invoked:

```
seg_srch_arg:  dli_operator  segment_name  L  G
                  {
                     .
                     :
                  concat  ") "  to abdl_str
                     .
                     :
                  }
               E  K
             |  dli_operator  E  K
```

We simply concatenate ") " to the abdl_str, and then the seg_srch_arg rule continues, by calling the E rule. The E rule searches for a GOTO statement in the input. Since no GOTO statement exists in this DL/I call, the empty portion of the E rule is matched, satisfying the E rule. Control returns to the seg_srch_arg rule, which then calls the K rule. The K rule searches for a dli_op in the input, i.e., the reserved word DLET or REPL. Since no such operator exists in this DL/I call, the empty portion of the K rule is matched, satisfying the K rule.

Next, control reverts to the seg_srch_arg rule, which is now fully satisfied. Therefore, control returns to

the ssa rule, which also is now fully satisfied. Then, control returns to the dml_statement rule, and the following SSL is invoked:

```
dml_statement: J  ssa
                  {
                    .
                    :
                  concat all attrs to last RETRIEVE req
                  concat ") BY 'seq-fld' ]" to last req
                    .
                    :
                  }
```

We now access the schema, again via the curr_seg_ptr, and obtain all the attributes for the COURSE segment, and concatenate them to the abdl_str. Of course, they are separated by commas. Next we concatenate ") BY " to the abdl_str. Then we access our only tgt_list item (where we previously stored the sequence field of the COURSE segment), and concatenate "CNUM]" to the abdl_str.

Now, the dml_statement is fully satisfied and control returns to the start statement that called it. The parsing and translation processes are now complete.

g. Segment Search Argument Command Codes

A segment search argument (SSA) may optionally include a command code. Command codes are special codes which allow variations to the basic DL/I calls. Each command code is represented by a single alphabetic character. Command codes are specified by writing an

asterisk, followed by the appropriate character, immediately after the segment name in the SSA.

(1) Path Retrieval (Command Code D). Normal GET operations retrieve data only for the segment type specified in the last SSA of the DL/I call. When command code D is included in an SSA, in connection with a GET operation, the effect is to retrieve data for the segment type specified in that SSA. In general, the D command code may be specified at some levels and not at others. The effect is to retrieve just the indicated segments. Of course, it is not necessary to specify the D command code in the final SSA, since this segment is retrieved by definition. An example of such a call is as follows:

```
GU  course  *D
    offering
    student
```

The series of ABDL requests generated for such a call is as follows:

```
[ RETRIEVE  (TEMPLATE = COURSE)
            (CNUM, CTITLE, DESCRIPN) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****))
            (DATE) BY DATE ]

[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (CNUM = ****) and
             (DATE = ******))
            (SNUM, SNAME, GRADE) BY SNUM ]
```

The only difference between these requests and those that would be generated without specifying command code D, is that the target lists for those SSAs specifying command code D include all the attributes of the segment type, instead of merely the sequence-field attribute. Those segment types specifying command code D are to be returned to the user.

(2) Path Insertion (Command Code D). Normal ISRT operations insert data only for the segment type specified in the last SSA of the ISRT call. Clearly, the parent and grandparent segments for such a segment type have to already exist within the database. With the specification of command code D in the DL/I ISRT, multiple segments may be inserted to the database in a single call. However, the segment types to be inserted still have to form an appropriate path that is consistent with the logical structure of the database, i.e., the structure defined by the hierarchical schema. Therefore, a parent segment is inserted; its child may be inserted next, since its parent now exists; and similarly for all other SSAs. The command code D specification is required only in the first SSA of the DL/I ISRT. An example of such a call is as follows:

```
build  (cnum) : ('cs69')
build  (date, location) : (850430, 'monterey')
build  (tnum, tname) : (1234, 'hsiao')
isrt   course  *D
       offering
       teacher
```

Notice that it is necessary for the user to build one segment for each SSA of the call, i.e., one segment for each segment type to be inserted. The series of ABDL requests generated for such a call is as follows:

```
[ INSERT (<TEMPLATE, COURSE>,
          <CNUM, Cs69>,
          <CTITLE, Zz>,
          <DESCRIPN, Zz>) ]

[ INSERT (<TEMPLATE, OFFERING>,
          <CNUM, Cs69>,
          <DATE, 850430>,
          <LOCATION, Monterey>,
          <FORMAT, Zz>) ]

[ INSERT (<TEMPLATE, TEACHER>,
          <CNUM, Cs69>,
          <DATE, 850430>,
          <TNUM, 1234>,
          <TNAME, Hsiao>) ]
```

One ABDL INSERT is generated for each SSA in the DL/I ISRT. Notice that no ABDL RETRIEVE requests are generated, since by definition, the sequence-field values have to be input by the user when building each new segment. These sequence-field values are saved by the KMS in local variables, so that they may be carried along, from segment to segment, as the translator successively generates each ABDL INSERT request. Also notice that three attribute values, not entered by the user when building the segments, have been defaulted to the value "Zz".

(3) Command Code F. Command code F provides a means of stepping backwards under the parent segment type that has been established as the current position within the database. As such, it is only specified when performing a GN, or GNP, DL/I GET call. As far as the KMS translator is concerned, there is no difference between such a call and a normal GN, or GNP. The translator generates the same series of ABDL requests in both cases. However, it does set a command-code flag that allows the KC to identify the functionality of these requests. An example using command code F is as follows, where we assume that the current position within the database has been established by one of the following sequences of DL/I calls:

```
GU  course                GU  course
    offering              GNP offering
GNP student
```

Then, corresponding subsequent calls that may be made by the user, specifying command code F, are as follows:

```
GNP teacher *F          GNP prereq *F
```

Command code F is disregarded if it is used at the root segment level (i.e., the root has no parent to step backwards under), or with a DL/I GU call.

(4) Command Code V. DL/I GNP calls only retrieve segments of the current parent type, as established by the previous DL/I call. By using the V command code, any ancestor may be designated as the current parent type. Therefore, the following sequences of DL/I calls retrieve identical student records:

```
GU  course                GU  course
    offering                  offering
GNP student               GN  offering  *V
                              student
```

Similarly, the following sequences of DL/I calls retrieve identical prerequisite records:

```
                          GU  course
GU  course                    offering
GNP prereq                GN  course    *V
                              prereq
```

The ABDL requests generated for such calls are no different than for similar requests not specifying the V command code. Again, the command-code flag is set by the translator to allow the KC to identify the functionality of these requests. The V command code may not be used with the last SSA of the call, nor may it be used in an SSA that includes occurrence qualification conditions, i.e., boolean predicates following the segment name.

3. Semantic Analysis

When the user desires to process DL/I requests against an existing database, the KMS first forms the equivalent ABDL requests. Then the KMS performs a semantic analysis of these current ABDL requests, relative to those requests generated during the previous call(s) to the KMS. The current ABDL requests are then integrated with those requests generated for the previous call, in a manner that depends upon the outcome of the semantic analysis.

In general, semantic analysis is only required when the current DL/I call is of the GN or GNP variety. Since these operations do not require the user to specify the complete hierarchical path, from the root to the desired segment, they have to be semantically analyzed relative to the previous DL/I GET operation (of any type) or ISRT operation. Specifically, the segment type referenced in the first SSA of the GN or GNP has to either : (1) match one of the SSAs in the previous DL/I call, in which case the two requests overlap, or (2) be the next segment type in the hierarchical path that logically follows the last SSA of the previous DL/I call, in which case the current call is a continuation of the previous call.

C. FACILITIES NOT PROVIDED BY THE IMPLEMENTATION

Our original intent has been to demonstrate that the hierarchical interface could indeed be developed and implemented. There are some facilities of DL/I that are not included in our implementation. Some of these facilities have more to do with providing an environment for the running of batch applications, than with supporting a germane hierarchical interface. For others, the programming time and effort required to incorporate them would be too costly for the benefits derived. However, this is not to imply that such facilities would not be useful. This section is devoted to describing the most prominent features of DL/I that are not supported by the language interface.

1. Interfacing the User

In our hierarchical interface, there is no concept of types of logical database records (LDBRs). An LDBR type may be thought of as a hierarchical arrangement of segment types derived from the underlying physical database record (PDBR) hierarchy. Any segment type of the PDBR hierarchy may be omitted from the LDBR hierarchy, and the attributes of an LDBR segment type may be a subset of those of the corresponding PDBR segment type. However, under our implementation, the logical database and the physical database are one in the same. Therefore, our interface is limited to data definition language (DDL) and data manipulation language (DML) statements, and provides no data

control facilities such as the SENSEG (sensitive segment) specification, the program communication block (PCB), or the PROCOPT (processing options) specification.

As mentioned in Chapter II, our interface data structures have been constructed to facilitate future use by multiple users. This would allow the LDBR concept to be supported by incorporating the hierarchical database schemas into the user information structure (user_info shown in Figure 8). These schemas would be logically external and user-specific with respect to the entire list of physical database schemas that are still global.

2. Segment Insertion Based on Current Position

A normal DL/I ISRT call specifies the complete hierarchical path from the root segment to the segment type being inserted. Although not included in our language interface, it is possible to omit the specification of the complete hierarchical path and to quote just the type of the new segment. In such a case, the current position within the database, that has been established during the previous call to the KC, is used to determine where the new segment is inserted.

This option, although not planned for during the design, is supported by the KMS parser/translator. However, as in the case of the GN or GNP, the specification of an incomplete hierarchical path makes it necessary to semantically analyze the ISRT SSAs and the previous DL/I

call. We feel the programming effort involved to go back and provide such a facility, although not complex, is time-consuming for the benefits to be derived.

3. Additional SSA Command Codes

The language interface supports the use of command codes D, F and V, as described earlier. These command codes are probably the most useful of the set of available DL/I command codes. For details of the remaining command codes (L, N, Q, U), see [Ref. 9: pp. 4.1-4.3].

The remaining command codes have not been implemented because we feel the effort involved to be too time-consuming to justify their benefits. Almost any DL/I operation that may be accomplished using these command codes may be done in an alternate fashion in our language interface, as it is presently implemented. A possible exception to this is command code Q, which concerns a data security concept that is beyond the scope of our present implementation.

V. THE KERNEL CONTROLLER

The Kernel Controller (KC) is the third module in the DL/I language interface and is called by the language interface layer (LIL) when a new database is being created or when an existing database is being manipulated. In either case, the LIL first calls the Kernel Mapping System (KMS) which performs the necessary DL/I-to-ABDL translations. The KC is then called to perform the task of controlling the submission of the ABDL transaction(s) to the multi-backend database system (MBDS) for processing. If the transaction involves inserting, deleting or updating information in an existing database, control is returned to the LIL after MBDS processes the transaction. If the transaction involves a retrieval request (i.e., GU, GN, GNP), the KC sends the translated ABDL request to MBDS, receives the results back from MBDS, loads the results into the appropriate file buffer, and calls the Kernel Formatting System (KFS) to format and display the results to the user. The other retrieval types (i.e., GHU, GHN, GHNP) are processed similarly, but the KFS is not called. These retrievals are used only to establish a currency position within the hierarchical database.

These ideas may be best illustrated by examining the following example. Suppose the user issues the following DL/I request:

```
GU  course (ctitle = 'mlds')
    offering (date = 850430)
    student (grade = 'a')
```

This request is translated to the following series of ABDL requests:

```
[ RETRIEVE  ((TEMPLATE = COURSE) and
             (CTITLE = Mlds))
            (CNUM) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****) and
             (DATE = 850430))
            (DATE) BY DATE ]

[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (CNUM = ****) and
             (DATE = ******) and
             (GRADE = A))
            (SNUM, SNAME, GRADE) BY SNUM ]
```

The KC is now called to control the transmission of these requests to MBDS for execution. Generally, this is accomplished by forwarding the first RETRIEVE request to MBDS. Results are gathered and placed in a file buffer. Notice that the next RETRIEVE is not fully-formed. Therefore, it is necessary to replace the asterisks with a value that is extracted from the first RETRIEVE request's

file buffer. In this example, the value is a course number (CNUM). Again, the request is forwarded to MBDS, and appropriate results are obtained. The last RETRIEVE request is also not fully-formed. In this case, attribute values from both the first and second RETRIEVE requests are utilized to complete the ABDL request. Thus, a value is pulled from the file buffer associated with the second request, and the same CNUM used to build the second request is again used to form the final request. The fact that a new value is not pulled from the first request's file buffer illustrates currency within the hierarchical database. Specifically, the values that are used in subsequent RETRIEVE requests have to be consistent with those values used in earlier requests. This ensures that the path used to retrieve values from the database is consistent with previous retrievals and the database hierarchy.

The procedures that make up the interface to the KDS (i.e., MBDS) are contained in the test interface (TI) of MBDS. To fully integrate the KC with the KDS, the KC calls procedures which are defined in the TI. Due to upcoming hardware changes in MBDS, we decided not to test the KC on-line with the TI. Our solution to this problem has been to design the system exactly as if it were interfacing with the TI. However, for each call to a TI procedure, we have created a software stub that performs the same functions as the actual TI procedure. The reader should realize that all

interactions with the TI procedures described in the KC are actually made with these software stubs, rather than with the on-line TI procedures.

In this chapter we discuss the processes performed by the KC. This discussion is in two parts. First, we examine the data structures relevant to the KC, followed by an examination of the functions and procedures found in the KC. Appendix D contains the design of our KC implementation, written in a system specification language (SSL).

A. THE KC DATA STRUCTURES

In this section we review some of the data structures discussed in Chapter II, focusing on those structures that are accessed and used by the KC. The first data structure used by the KC is the dli_info record shown in Figure 18. The KC makes use of only two fields in this record. The first, curr_sit_pos, is a pointer to an Sit_status_info record, shown in Figure 19. This record indicates to the KC at what location in the list of Sit_info nodes execution is to begin. The second field of interest, buff_count, is an integer used to maintain control of the file buffers associated with the results of each RETRIEVE request. For instance, the results associated with the first RETRIEVE request of our last example are placed in a file buffer with an extension of "0". The buff_count is incremented by one,

```
struct  dli_info
  {
    struct  curr_db_info      curr_db;
    struct  file_info         file;
    struct  tran_info         dli_tran;
    struct  ddl_info          *ddl_files;
    int                       answer;
    int                       operation;
    int                       error;
    union   kms_info          kms_data;
    struct  Sit_info          *sit_list;
    struct  Sit_info          *kms_sit;
    struct  hrec_node         *saved_seg_ptr;
    struct  hrec_node         *saved_seg_ptr2;
    struct  Sit_status_info   *fst_sit_pos;
    struct  Sit_status_info   *curr_sit_pos;
    int                       buff_count;
  }
```

Figure 18. The dli_info Data Structure.

and the results associated with the second request are placed in a file buffer with an extension of "1".

As noted above, the Sit_status_info record indicates to the KC where execution of a group of ABDL requests is to begin. (See Figure 19.) The first field, req_pos, is a pointer to an Sit_info record, which holds the information

```
struct  Sit_status_info
  {
    struct  Sit_info          *req_pos;
    struct  Sit_status_info   *next;
    int                       status;
  }
```

Figure 19. The Sit_status_info Data Structure.

required by the KC to properly control the execution of the request. The following field, next, is a pointer to the next Sit_status_info node that the KC is to process. This field may be NULL if no other requests are to be processed. The last field, status, is an integer which indicates how much of the current request overlaps the previous request. For example, if the DL/I request shown in our first example is followed by:

```
GN  offering (date = 850430)
    student (grade = 'a')
```

then the status field would indicate that this request overlaps our first request at the OFFERING and STUDENT segments. There may also be no overlap between requests. For instance, if our example database (see Figure 3) contained GRADUATE and UNDERGRADUATE segments below the STUDENT segment, and if our first example is followed by:

```
GN  graduate (gname = 'jones')
```

then there is no overlap.

The Sit_info record, shown in Figure 20, contains the information needed by the KC to process a DL/I request. The first two fields, prev and next, are pointers to the previous and next Sit_info nodes, respectively, and are used by the KC to obtain information about the previous and next

Sit_info nodes. The third field, parent, is also a pointer to an Sit_info node. However, in this case the pointer is to the parent node. Information about the parent node is required by the delete and special-retrieve procedures for proper execution. The next two fields, child and sibling, are pointers to child and sibling Sit_info nodes. They are also used to process deletes and special-retrieves. The reader should note that these fields effectively represent the hierarchical form of the database, although the nodes are physically stored as a linearly-linked list. The following field, loop, is also a pointer to an Sit_info node, but is used to indicate where the KC should loop when

```
struct  Sit_info
  {
    struct  Sit_info        *prev;
    struct  Sit_info        *next;
    struct  Sit_info        *parent;
    struct  Sit_info        *child;
    struct  Sit_info        *sibling;
    struct  Sit_info        *loop;
    struct  Sit_info        *nf_loop;
    char                    *abdl_req;
    char                    *template;
    int                     operation;
    int                     cmd_code;
    int                     or;
    char                    seg_name[RNLength + 1];
    int                     BOR;
    int                     EOR;
    struct  hie_file_info   *result_file;
  }
```

Figure 20. The Sit_info Data Structure.

a GOTO is encountered in the DL/I request. For example, suppose the user issues the following DL/I request:

```
        GU   course (ctitle = 'mlds')
             offering (date = 850430)
xx      GN   student (grade = 'a')
        GOTO xx
```

This request retrieves all students receiving a grade of "A" in the course entitled "mlds", that is offered on 850430. We have seen that without the GOTO, this request is translated to three ABDL RETRIEVE requests. Since we desire to retrieve all STUDENT segments for the above request, it is necessary to provide a pointer to the Sit_info node that we may loop on. In this case, it is the Sit_info node associated with the retrieval of the STUDENT segments, i.e., the last RETRIEVE shown in our first example.

The next two fields, abdl_req and template, are pointers to character strings. The first, abdl_req, holds the ABDL request previously parsed by the KMS. This array may contain place-holding asterisks if the request is not fully-formed. The template field is used to build a fully-formed ABDL request. Thus, it never contains asterisks. These have been substituted with appropriate values from the file buffers. The reader may ask why the fully-formed ABDL request is not built on top of the abdl_req field? The problem with this is that abdl_req may be used in subsequent

ABDL actions with different values being substituted for the asterisks with each new action. If the place-holding asterisks are destroyed, then there is no way to determine where to place the new values in the request. The following field, operation, is an integer indicating the DL/I operation associated with this Sit_info node, i.e., GU, GN, GNP, DLET, ISRT, GHU, GHN, GHNP, SPECRET. (Here, we use a SPECRET operation code to refer to GN and GNP requests with no SSAs.) The KC uses this information to invoke the correct procedure to execute the ABDL equivalents of the DL/I request. The next field, cmd_code, is a flag set by the KMS to indicate the presence of a particular SSA command code in the DL/I request.

The following field, or, is a flag that indicates if an "or" is present in an ABDL request. For example, the KMS sets this field to TRUE as a result of the "or" between the dates in the OFFERING SSA of the following DL/I request:

```
GU  course
    offering (date = 840430  or  date = 850430)
```

The KC needs to know this information when it builds a request for subsequent execution. If the above request is issued, its translation is as follows:

```
[ RETRIEVE  (TEMPLATE = COURSE)
            (CNUM) BY CNUM ]

[ RETRIEVE  (((TEMPLATE = OFFERING) and
              (CNUM = ****) and
              (DATE = 840430))
          or ((TEMPLATE = OFFERING) and
              (CNUM = ****) and
              (DATE = 850430)))
             (DATE, LOCATION, FORMAT) BY DATE ]
```

Because of the "or", the same course number has to be used in both instances of the asterisks in the second RETRIEVE. The or field is used to signal the KC when this occurs.

The next field, seg_name, holds the segment name specified in the SSA of the DL/I request. The following two fields, BOR and EOR, serve as flags indicating the beginning and end of a request. If we use our last example, BOR for the first RETRIEVE request is set to TRUE, while EOR for the second (last) RETRIEVE is set to TRUE. These values are used to control the execution of ABDL requests. For instance, the KC may continue to execute RETRIEVEs until it detects a TRUE value in the EOR field.

The last field, result_file, is a pointer to the hie_file_info record, shown in Figure 21. This record stores information about file buffers containing the results obtained for each RETRIEVE request. The first field, buff, contains the file name and file id. This information is required so that the appropriate files may be written to, read from and appended to, as necessary. The second field,

```
struct  hie_file_info
  {
    struct  file_info  buff;
    int                count;
    int                status;
    int                buff_loc;
    char               *curr_buff_val;
  }
```

Figure 21. The hie_file_info Data Structure.

count, is simply an integer representing the number of results in the file buffer. The next field, status, serves as a flag so that a file buffer is opened under the correct status. The fourth field, buff_loc, indicates the KC's location in the file buffer. For instance, after the first value is pulled from a file buffer, this field indicates that the KC's position is now at the beginning of the second result. The last field is a pointer to a character string and holds the last result value pulled from the file buffer. This field is used to maintain a currency position in the database hierarchy. Once a value is obtained from the file buffer, it is difficult to reset the file pointer to the location where the value has just been obtained. It is easier to simply store the value so that it may be used in the building of subsequent RETRIEVE requests.

B. FUNCTIONS AND PROCEDURES

The KC makes use of a number of different functions and procedures to manage the transmission of the translated DL/I requests (i.e., ABDL requests) to the KDS. Not all of these functions and procedures are discussed in detail. Instead, we provide the reader with an overview of how the KC controls the submission of the ABDL requests to MBDS.

1. The Kernel Controller

The dli_kc procedure is called whenever the LIL has an ABDL transaction for the KC to process. This procedure provides the master control over all other procedures used in the KC. The first portion of this procedure initializes global pointers that are used throughout the KC. Specifically, kc_curr_pos is set to point to the first Sit_info node that is to be processed by the KC, and kc_ptr is set to the address of the li_dli structure for a particular user. The remainder of this procedure is a case statement that calls different procedures based upon the type of ABDL transaction being processed. If a new database is being created, the load_tables procedure is called. If the transaction is of any other type, then requests_handler is called. If the transaction is none of the above, there is an error and an error message is generated with control returned to the LIL.

2. Creating a New Database

The creation of a new database is the least difficult transaction that the KC handles. The load_tables procedure is called, which performs two functions. First, the test interface (TI) dbl_template procedure is called. This procedure is used to load the database-template file created by the KMS. Next, the TI dbl_dir_tbls procedure is called. This procedure loads the database-descriptor file. These two files represent the attribute-based metadata that is loaded into the KDS, i.e., MBDS. After execution of these two procedures, control returns to the LIL.

3. The GU, GN, GNP, ISRT and REPL Requests

The GU, GN, GNP, ISRT and REPL requests are all handled in a similar manner. For any one of these types of operations, the GU_proc procedure is called. The following examples illustrate the logic used in this procedure which controls the processing of these types of requests. Suppose the following DL/I request is issued by the user:

```
GU  course
    offering (date = 850430)
    student (grade = 'a')
```

The KMS translates this DL/I request into the following three ABDL RETRIEVE requests:

```
[ RETRIEVE   (TEMPLATE = COURSE)
             (CNUM) BY CNUM ]

[ RETRIEVE   ((TEMPLATE = OFFERING) and
              (CNUM = ****) and
              (DATE = 850430))
             (DATE) BY DATE ]

[ RETRIEVE   ((TEMPLATE = STUDENT) and
              (CNUM = ****) and
              (DATE = ******) and
              (GRADE = A))
             (SNUM, SNAME, GRADE) BY SNUM ]
```

Also suppose this is the first request the user issues against the database. The kc_curr_pos is set to point to the first ABDL RETRIEVE request shown above. In addition, the fst_sit_pos of dli_info indirectly points to the first RETRIEVE. Therefore, the first task GU_proc accomplishes is to determine if kc_curr_pos and fst_sit_pos point to the same Sit_info node. If they do, then the KC knows that this is the first request issued by the user, and that the first RETRIEVE request is fully-formed (the case where the two fields are not the same is examined in our next example).

Since the first RETRIEVE request is complete, it may be immediately forwarded to the KDS for execution. This is accomplished by calling dli_execute. This procedure uses two TI procedures and the dli_chk_requests_left procedure. In general, dli_execute sends the ABDL request to the KDS and waits for the last response to be returned. Results for a given request are placed in a unique file buffer

associated with each Sit_info node. The file_results procedure controls this process.

After the last response is returned, control is returned to GU_proc. Now, GU_proc has to process the remaining RETRIEVE requests until the end-of-request flag is detected. Therefore, kc_curr_pos now points to kc_curr_pos->next, which is in this case, the second RETRIEVE. However, this RETRIEVE request may not be forwarded to the KDS because it is incomplete. Hence, the build_request procedure is called to complete the request. In this instance, a course number (CNUM) is substituted for the place-holding asterisks. This value is obtained from the first RETRIEVE's file buffer. Specifically, this value is located in curr_buff_val of the first RETRIEVE's result_file. This RETRIEVE may now be forwarded to the KDS for execution in the same fashion as the first RETRIEVE.

Finally, the last RETRIEVE request has to be processed. Again kc_curr_pos is set to point to kc_curr_pos->next, i.e., the last RETRIEVE. This RETRIEVE request is also incomplete, so build_request is called to complete the request. However, a value from both the first and second RETRIEVE's file buffer is used to complete the request. We note that the same course number used to build the second RETRIEVE is also used to build the last RETRIEVE. This is because we have established a currency position within the database that is related to the first value in

the first RETRIEVE's file buffer and the first value in the second RETRIEVE's file buffer. As before, this request is forwarded to the KDS for execution once it is fully-formed.

If results are returned, then dli_kfs is called to display to the user the first STUDENT segment satisfying the request. If, on the other hand, results are not returned, then the KC has to retract a level in the hierarchy, obtain the next value from that level's file buffer, and re-issue the request to the KDS. In this example, the KC would retract to the level of the second RETRIEVE, pull the second value from its file buffer, substitute this value for the asterisks related to the date in the last request, and again forward the request to the KDS.

Let's look at an extreme instance where the KC is unable to obtain any STUDENT segments for any of the values in the second RETRIEVE's file buffer. In this case, it would be necessary to retract all the way to the level of the first RETRIEVE, pull its second value from the file buffer, substitute it for the asterisks in the second RETRIEVE, and re-issue the request to the KDS for execution. This process continues until either a STUDENT segment is returned, or the KC uses the last value in the first RETRIEVE's file buffer and no STUDENT segment is returned. This would indicate that no STUDENT segments exist in the database for this particular request.

Suppose now that the DL/I request we have just discussed is followed by:

```
yy   GN  student (grade = 'a')
     GOTO  yy
```

The KMS translation of this DL/I request is as follows:

```
[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (CNUM = ****) and
             (DATE = ******) and
             (GRADE = A))
            (SNUM, SNAME, GRADE) BY SNUM ]
```

This request is linked to the last RETRIEVE of the previous example, and it is both a beginning-of-request and an end-of-request. The KC is again called with kc_curr_pos now pointing to the above RETRIEVE. The KC recognizes this request as a GN operation, therefore, GU_proc is called. However, this time kc_curr_pos and fst_sit_pos do not point to the same Sit_info node. Thus, this is not the first request issued by the user. Therefore, subsequent action taken by the KC is based on the status field in the Sit_status_info record, set during the semantic analysis in the KMS. If the status field is set to MATCHALL (indicating SSA overlap between this, and the previous DL/I request), then the KC determines if all values in the file buffer have been returned. If they have, then it is necessary to

retract to the next higher level and try to re-issue the request. However, if all values in the file buffer have not been returned, then dli_kfs is called to display the next value in the file buffer. In our example, the status field is set to MATCHALL. Thus, the actions described above are taken.

However, now suppose that our first example DL/I request had been:

```
GU  course
    offering (date = 850430)
```

followed by:

```
GN  student (grade = 'a')
```

In this instance, the status field is set to MATCHPART (indicating the SSAs of this request are a continuation of the previous DL/I request). The RETRIEVEs are identical. However, the KC has to process and execute the RETRIEVE request associated with "GN student..." before calling dli_kfs. This is because this RETRIEVE has not been executed, as it had been when the status field had been set to MATCHALL.

Let's return to our first example where the DL/I request:

```
        GU  course
            offering (date = 850430)
            student (grade = 'a')
```

is followed by:

```
        yy  GN  student (grade = 'a')
            GOTO  yy
```

There is a GOTO in the second DL/I request, which means that the loop pointer is set to a value other than NULL. In this example, the loop pointer both emanates and points to the RETRIEVE associated with the "GN student..." request. Therefore, the loop_handler procedure is called to control the looping to this node and subsequent display of all STUDENT segments satisfying the request. Although our example does not show it, the loop pointer may point to an Sit_info node in the middle of a group of requests. In this case, the loop_handler procedure processes all RETRIEVEs from where the loop pointer points, to the end of the group of requests. This is done until all results in the file buffer pointed to by the loop pointer have been used.

The reader may notice that we have not discussed the other DL/I request operations that are processed by the GU_proc procedure. This is because the logic is the same whether the operation is a GU, GN, GNP, ISRT or REPL. The KC knows that it is receiving a linked-list of requests,

delimited by a beginning-of-request and an end-of-request. Therefore, the logic in GU_proc is predicated on detecting these flags and processing all requests in between, without regard to the specific operation.

4. The GHU, GHN, and GHNP Requests

The GHU, GHN, and GHNP requests are handled in a similar manner. The logic is exactly the same as that described in the last subsection for GU_proc. However, instead of calling dli_kfs to display a segment when the end-of-request is detected, control is returned to dli_kc for further processing of any additional requests. The intent of these operations is to establish a currency position within the database. This is done by moving the file-buffer pointer to the correct position within the buffer. Therefore, the procedure that processes these operations (i.e., GHU_proc) moves this pointer, instead of calling dli_kfs.

5. The DLET and SPECRET Requests

DLETs and SPECRETs (i.e., GN and GNP with no SSAs) are the most difficult operations for the KC to process. The problem with handling these operations is that they affect the entire database hierarchy as opposed to just a linear path within the database. This idea is illustrated by the following example. Suppose the user issues the following DL/I request:

```
GHU  course (ctitle = 'mlds')
     offering (date = 850430)
DLET
```

This request first retrieves all course numbers for which the course title is "mlds". This is followed by another RETRIEVE request that gathers all dates for a course number (retrieved above) and an offering date equal to 850430. These RETRIEVEs are used to gather the results needed to process the DLETs for this segment and all its children. (In this case, the appropriate TEACHER and STUDENT segments.) The KMS translation of this request is as follows:

```
[ RETRIEVE  ((TEMPLATE = COURSE) and
             (CTITLE = Mlds))
            (CNUM) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****) and
             (DATE = 850430))
            (DATE) BY DATE ]

[ DELETE ((TEMPLATE = OFFERING) and
          (CNUM = ****) and
          (DATE = ******)) ]

[ DELETE ((TEMPLATE = TEACHER) and
          (CNUM = ****) and
          (DATE = ******)) ]

[ DELETE ((TEMPLATE = STUDENT) and
          (CNUM = ****) and
          (DATE = ******)) ]
```

The reader may easily discern that we are not only deleting those records for which the course name is "mlds" and the offering date is 850430, but we are also deleting the children of any records for which these conditions are true. Our solution to this problem is the use of mutual recursion. Generally, when the KC detects that a DELETE operation is to be performed it calls the Delete_proc procedure. Delete_proc deletes records from the database until the kc_curr_pos pointer becomes NULL. During processing, Delete_proc determines if a node to be deleted has a child node. If the node does, then delete_setup is called. The delete_setup procedure then calls Delete_proc for all child records associated with this parent record. If we look at our example again, we see that the first DELETE node has a child, which is the DELETE-TEACHER node. This in turn has a sibling, which is the DELETE-STUDENT node. Therefore, it is necessary to delete all TEACHER and STUDENT segments that are children of the OFFERING segment. The deletion of these child segments is continued until the parent node's file buffer is exhausted. This is the file buffer of the DELETE-OFFERING node in our example.

The SPECRET operation works in a similar manner. This operation is required when all the children of a particular segment are also to be retrieved. Again, this occurs during GN and GNP DL/I requests when no SSAs are specified.

VI. THE KERNEL FORMATTING SYSTEM (KFS)

The KFS is the fourth module in the DL/I language interface, and is called by the Kernel Controller (KC) when it is necessary to display results to the user. The transformation of data into the appropriate format is a very simple task for the DL/I language interface. Unlike most other language interfaces, no change in format is required. The form that the data is in when it is retrieved from MBDS is the same form in which it is to be displayed to the user. The task of the KFS is reduced to simply printing out the results obtained from the ABDL equivalents of the DL/I requests. In this chapter, we discuss how the KC stores the data that the KFS eventually displays, and how the KFS outputs this data. Appendix E contains the design of the KFS, written in a system specification language (SSL).

A. THE KFS DATA STRUCTURE

The KFS utilizes just one of the data structures defined in the language interface. The kfs_hie_info record, shown in Figure 22, contains information needed by the KFS to process the results. The first field in this record, response, contains the result from MBDS which is loaded by the KC just prior to calling the KFS. The second field, curr_pos, lets the KFS know where it is in the response

```
struct  kfs_hie_info
  {
    char  *response;
    int   curr_pos;
    int   res_len;
  }
```

Figure 22. The kfs_hie_info Data Structure.

buffer. This assists the KFS in maintaining the correct orientation in the response buffer. The last field, res_len, indicates the length of the response buffer. This value is used as a halting condition. For instance, the KFS continues to pull characters out of the response buffer while the loop index is less than or equal to the res_len.

B. THE FILING OF DL/I RESULTS

The KC stores the results obtained from a DL/I request by calling the file_results procedure. This procedure first determines whether the response being returned by MBDS is the initial response to a DL/I request. If it is the initial response, then the result file is opened for writing in the response. If the incoming response is not the initial one, then the results file is opened for appending the new response to older responses. The procedure reads in the name of the first attribute and stores it in a variable, in addition to storing it in the results file. The attribute value is then stored into the results file. A while loop then handles the storing of the remaining

attribute-value pairs into the results file. Before an attribute name is stored into the results file, a check is made to determine if this attribute matches the attribute name of the segment sequence field. If the attribute names match, an end-of-line marker is inserted in the results file before the attribute-value pair is stored. Otherwise, the attribute-value pair is stored without the end-of-line marker. This check is one of the reasons that the KFS task of formatting output is so easy for the DL/I language interface.

C. THE KFS PROCESS

The KFS module is contained in the small procedure dli_kfs. The KFS is only called by the KC when the results of a request are to be displayed to the user. The only task that the KFS performs is to display to the screen the attribute-value pair found on the current line in the results file. A loop prints out this line, a character at a time, until the end-of-line or end-of-file marker is reached. The current position within the results file is then incremented by one and control is returned to the KC.

VII. CONCLUSION

In this thesis, we have presented the specification and implementation of a DL/I language interface. This is one of four language interfaces that the multi-lingual database system is to support. In other words, the multi-lingual database system is to be able to execute transactions written in four well-known and important data languages, namely, DL/I, SQL, CODASYL, and Daplex. DL/I is, of course, the well-known hierarchical data language provided by, for example, the IBM Information Management System (IMS). In our case, we support DL/I transactions with our language interface by way of the LIL, KMS, KC and KFS, in place of IMS. A related thesis by Kloepping and Mack [Ref. 19] examines the specification and implementation of the SQL interface. This work is part of ongoing research being conducted at the Laboratory of Database Systems Research, Naval Postgraduate School, Monterey, California.

The need to provide an alternative to the development of separate stand-alone database systems for specific data models has been the motivation for this research. In this regard, we have shown how a software DL/I language interface may be constructed. Specific contributions of this thesis include the development of useful algorithms and the

implementation of DL/I operations such as: sequential retrieval without SSAs, sequential retrieval without SSAs within a parent, command codes F and V, and path retrieval and path insertion (command code D). In addition, we have developed a LIL that is virtually reusable. With minor modifications the LIL may be used with the other language interfaces. Our generic data structure design is also noteworthy. Because of our extensive utilization of unions (i.e., variant records), the other language interfaces may use our generic data structures. We have extended the work of Banerjee [Ref. 2] and Weishar [Ref. 3] by specifying and implementing the algorithms for the language interface. In addition, we have also provided a general organizational description of the MLDS.

A major design goal has been to design a DL/I language interface to MBDS without requiring that changes be made to MBDS or ABDL. Our implementation is completely resident on a host computer. All DL/I transactions are performed in the DL/I interface. MBDS continues to receive and process transactions written in the unaltered syntax of ABDL. In addition, our implementation has not required any changes to the syntax of DL/I. We are implementing DL/I in an interactive language interface. However, DL/I is an embedded database language that is invoked from a host language (i.e., PL/I, COBOL, or System/370 Assembler Language) by means of subroutine calls. The syntax for

providing an attribute-value pair to be changed during a DL/I REPL call, and for building a new segment to be inserted during a DL/I ISRT call, is resident in the host language, not in the DL/I data language itself. In order to make such an embedded language function interactively, we have been forced to introduce additional syntax for the language interface. This additional syntax does not represent a change to the DL/I data language, but rather, serves only to facilitate our interactive implementation of the normally embedded data language, DL/I. The interface is completely transparent to the DL/I user.

In retrospect, our level-by-level top-down approach to designing the interface has been a fine choice. This implementation methodology had been the most familiar to us and proved to be relatively time efficient. In addition, this approach permits follow-on programmers to easily maintain and modify (when necessary) the code. Subsequent programmers will know exactly where we stopped because we made many of the lower-levels stubs. Hence, it is an easy task to replace these stubs with code. This is an advantage of this approach that we did not realize until completion of our implementation.

We have shown that a DL/I interface may be implemented as part of a MLDS. We have provided a software structure to facilitate this interface, and we have developed the actual code for implementation. The next step is to implement the

other interfaces. When these are complete, the system needs to be tested as a whole to determine how efficient, effective, and responsive it is to users' needs. The results may be the impetus for a new direction in database system research and development.

APPENDIX A - SCHEMATIC OF THE MLDS DATA STRUCTURES

The purpose of this appendix is to present a pictorial of the data structures used in the DL/I language interface. Since the code used for our thesis is the C programming language, the diagrams make use of its constructs, just as the code does. Groups of related items are known as structures in C, and it is easy to see from the diagrams that each structure breaks down into more detailed, workable structures. There are two major parts of this appendix. In Figure 23 we present the hierarchical database schema data structures that are discussed in Chapter II. In Figure 24 we present the user data structures.

In the diagrams, an arrow indicates that the field is a pointer to a structure. Each of the fields of such a structure is preceded by a small arrow to indicate that, indeed, a pointer from another structure is referencing the field. An example of this is the di_ddl_files field of the dli_info structure in Figure 24 on page 121. The field di_ddl_files points to a structure of type ddl_info. This convention is especially useful when writing or tracing long paths through the user data structure.

On the other hand, bracket lines are used to indicate when the field of a structure is also a structure. The bracket lines are drawn from the "parent" field to the "child" structure. A period is placed in front of the bracketed structure's fields to indicate this fact. An

example of this is the di_dli_tran field of the dli_info structure in Figure 24 on page 122. The field di_dli_tran is a structure of type tran_info. The bracket lines and the periods indicate this.

We note that the diagram has a few instances of UNIONS. A union is a construct that allows the user to connect different structure types, specified by the union structure, to a common structure, i.e., unions are also referred to as variant records. Since the multi-lingual database system is to support the mapping of multiple languages, many portions of the user structure are the same for any language used. However, the union construct allows for the parts that have to change between language interfaces, so that the common data structures may be adapted to be useful to all language interfaces.

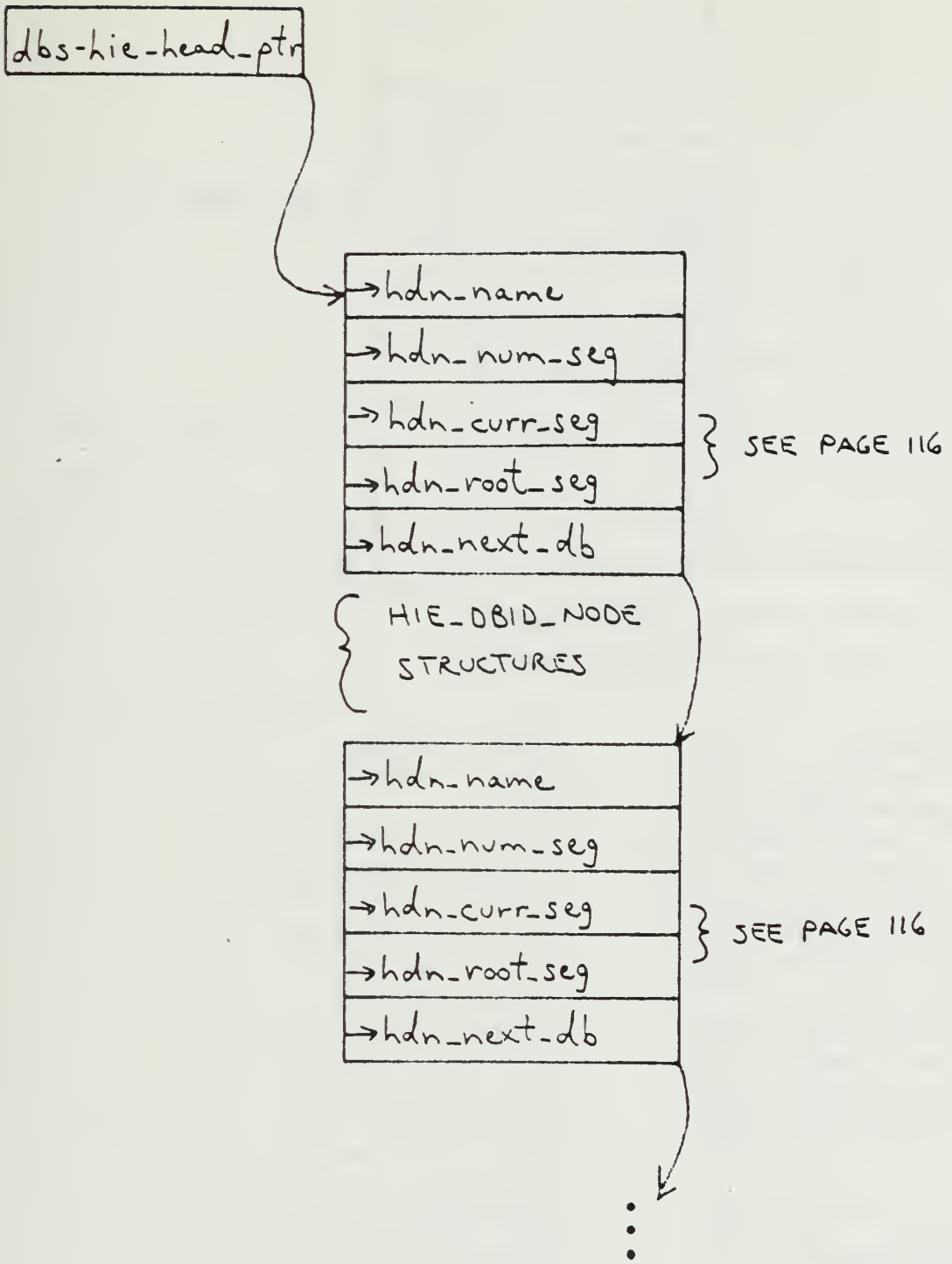


Figure 23. Hierarchical Database Schema Data Structures

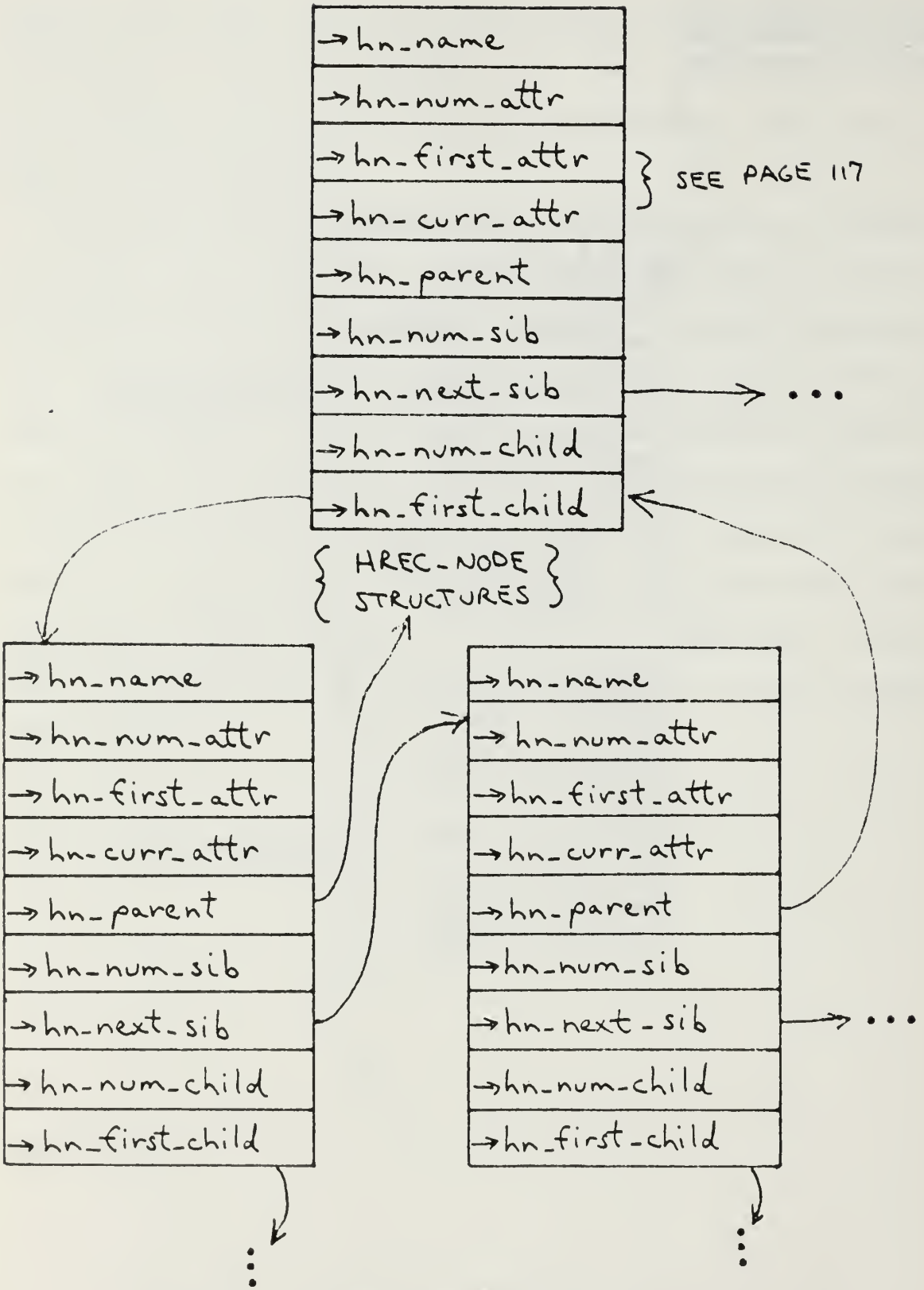


Figure 23. Continued

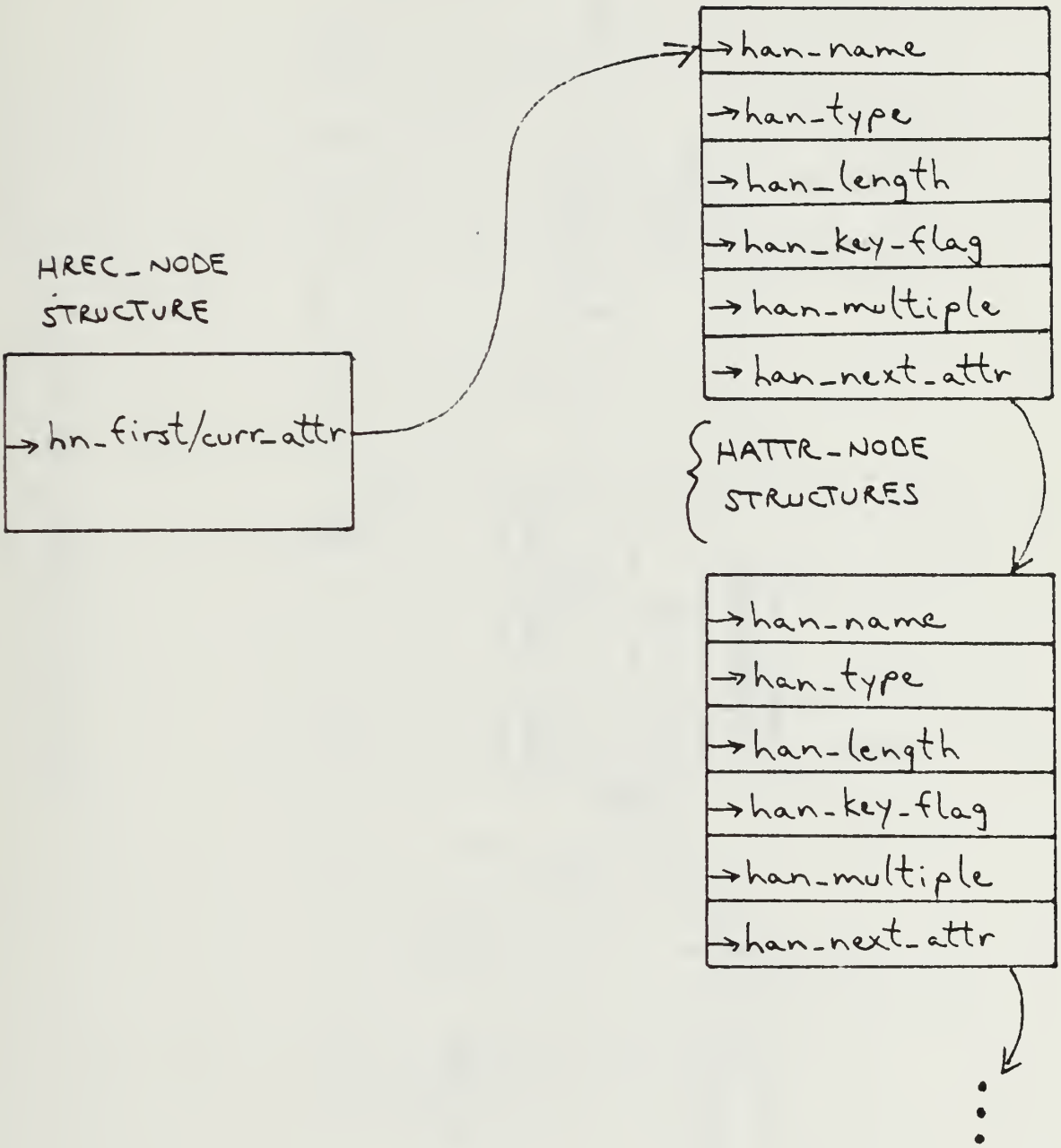


Figure 23. Continued

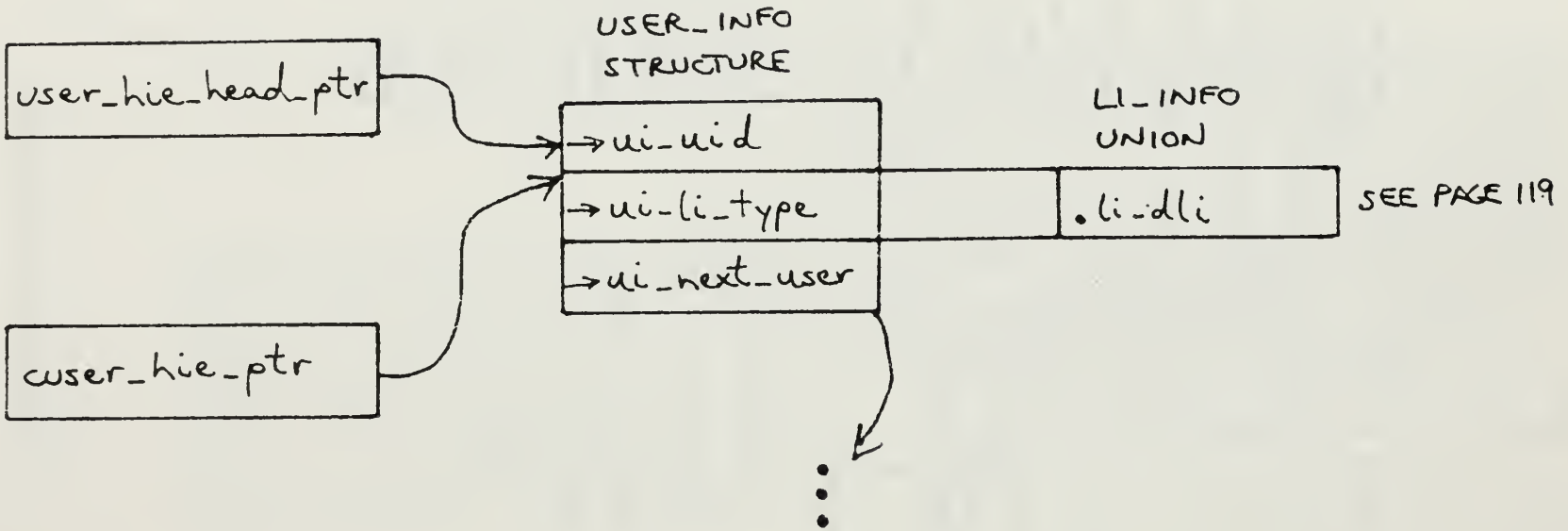


Figure 24. User Data Structures

DLI_INFO
STRUCTURE

.di_curr_db	SEE PAGE 120
.di_file	SEE PAGE 120
.di_dli_tran	SEE PAGE 122
.di_ddl_files	SEE PAGE 121
.di_answer	
.di_operation	
.di_error	
.di_kms_data	SEE PAGE 123
.di_kfs_data	SEE PAGE 127
.di_kc_data	
.di_sit_list	SEE PAGE 128
.di_kms_sit	SEE PAGE 128
.di_saved_seg_ptr	
.di_saved_seg_ptr2	
.di_fst_sit_pos	SEE PAGE 129
.di_curr_sit_pos	SEE PAGE 129
.di_buff_count	

Figure 24. Continued

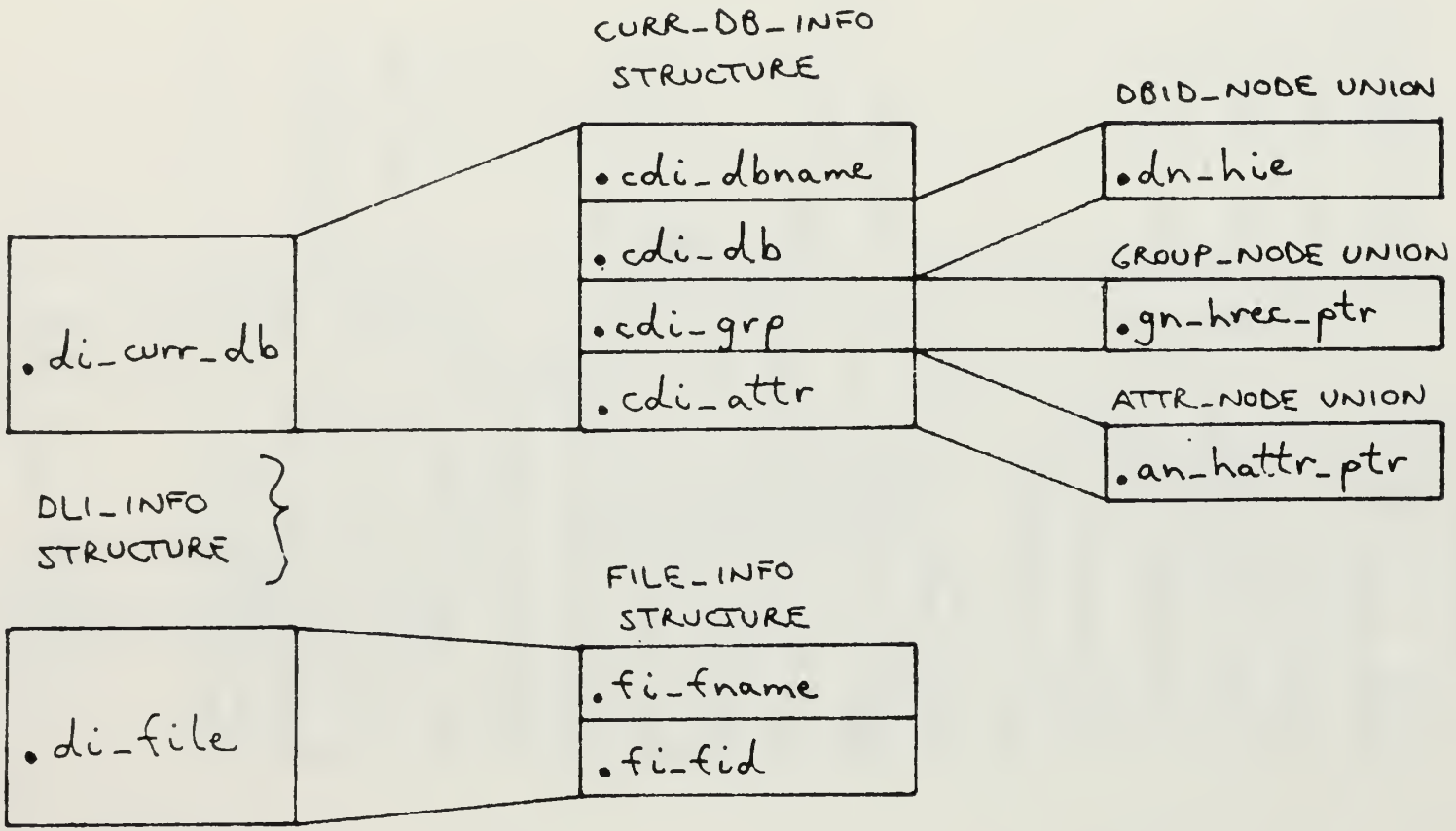


Figure 24. Continued

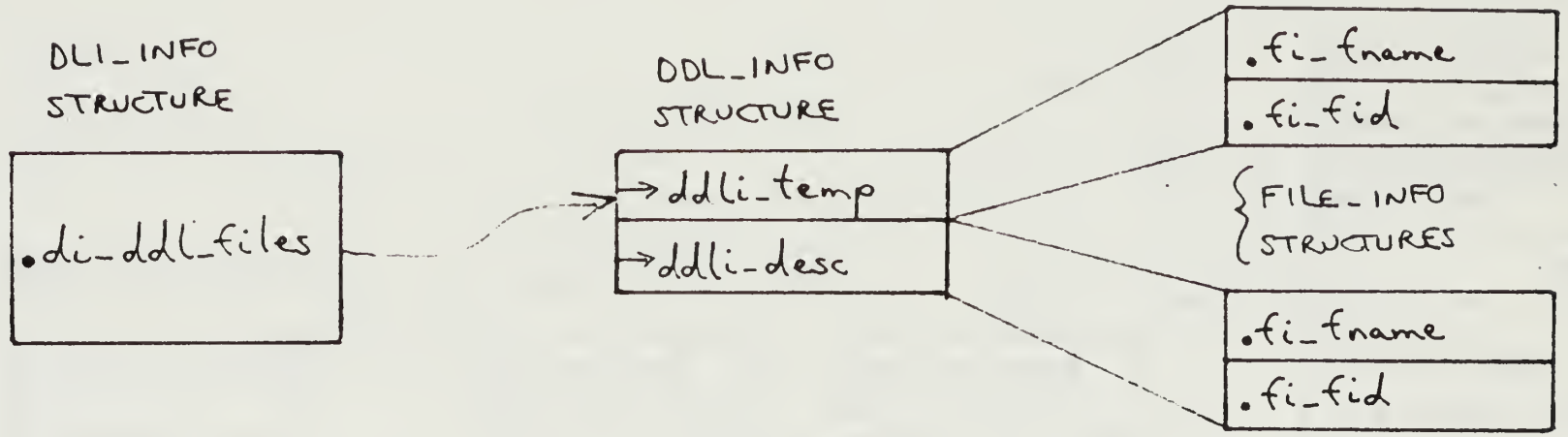


Figure 24. Continued

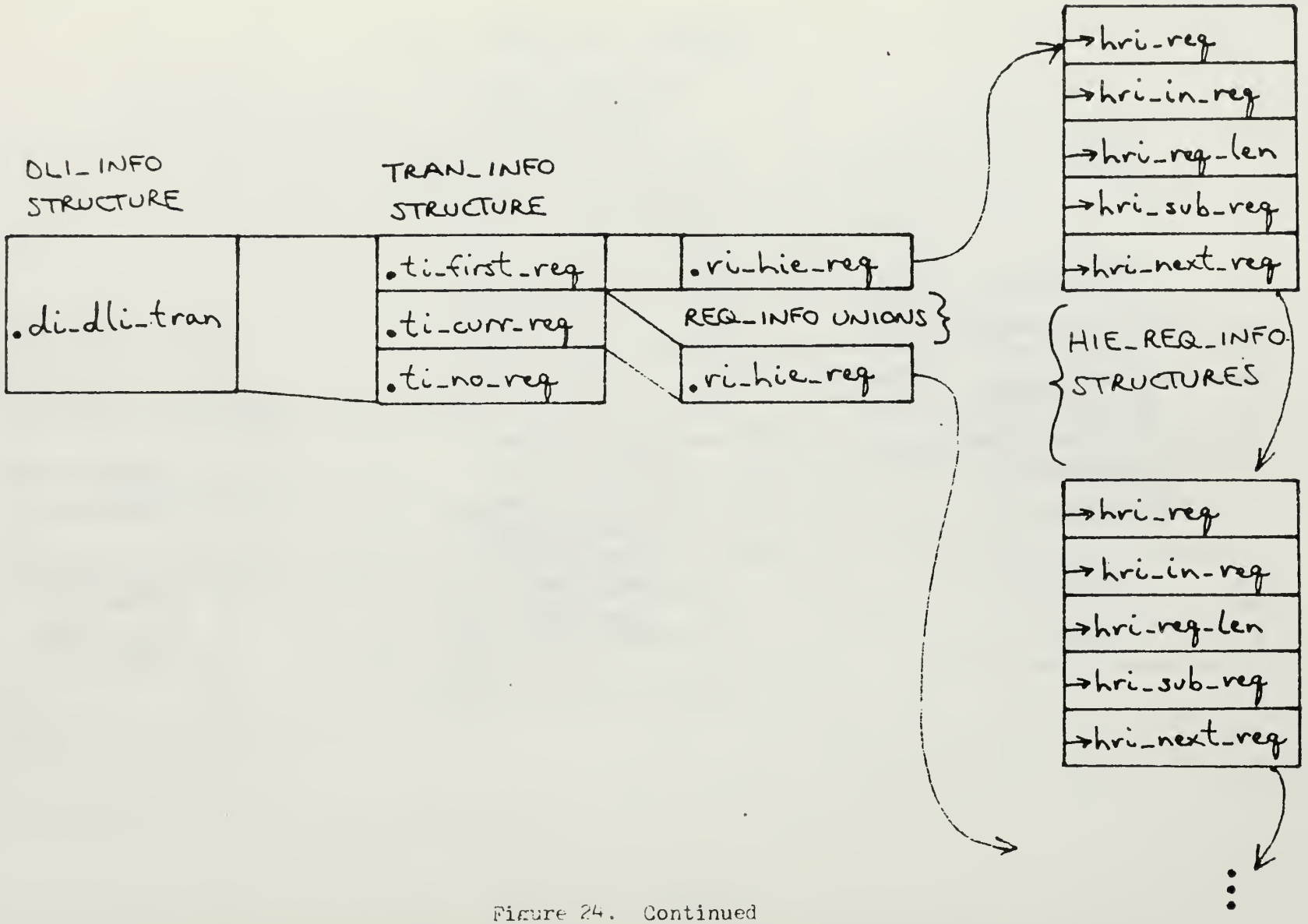


Figure 24. Continued

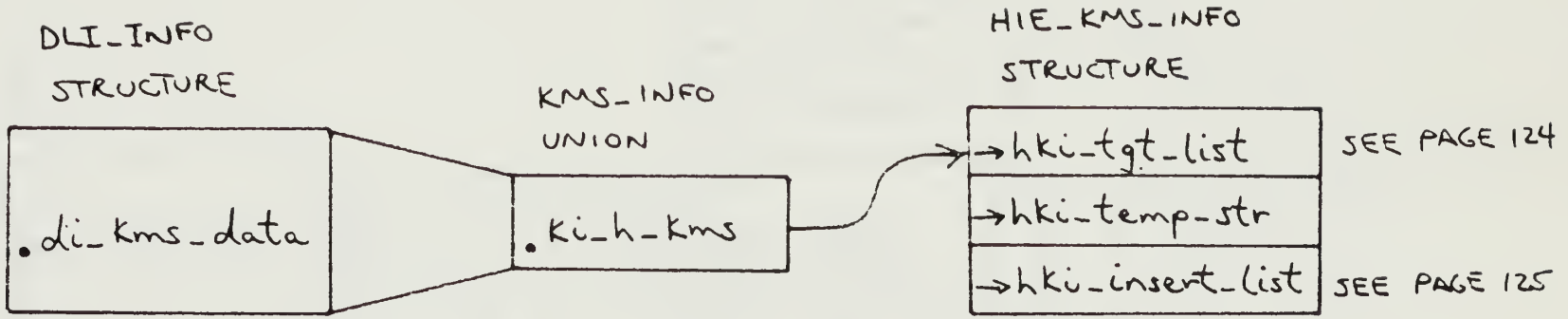


Figure 24. Continued

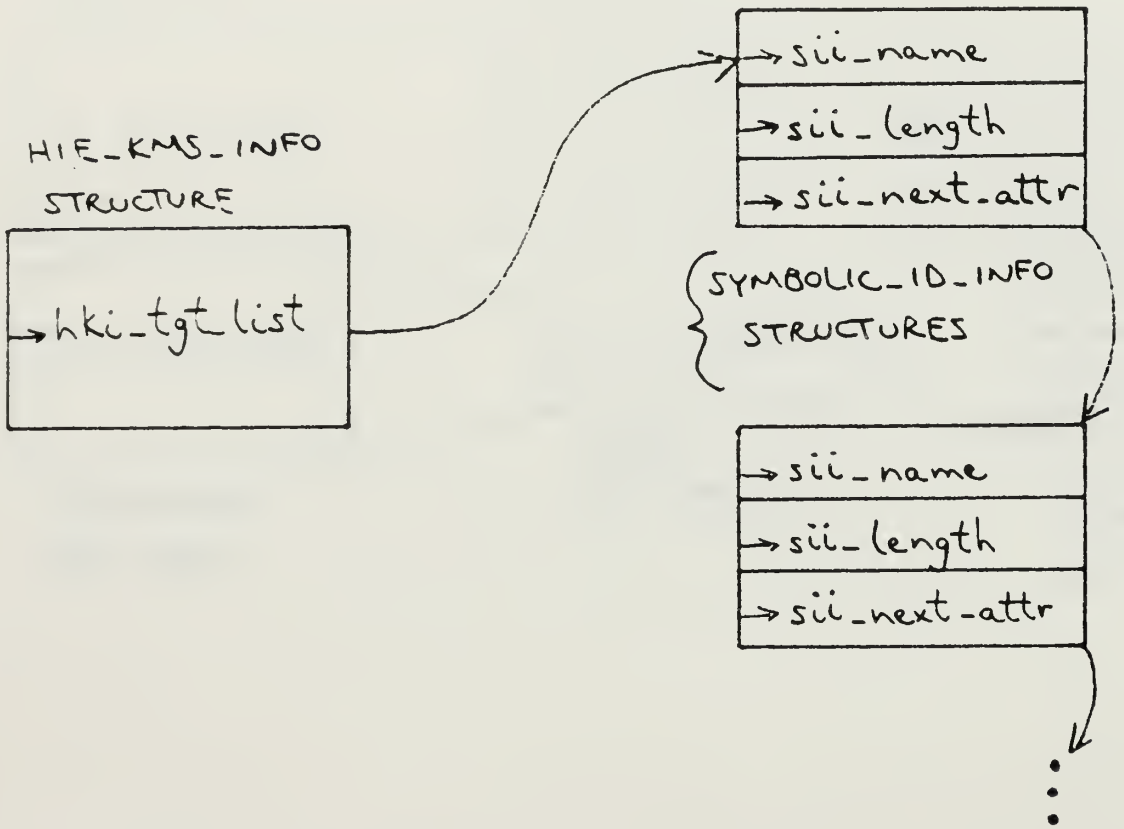


Figure 24. Continued

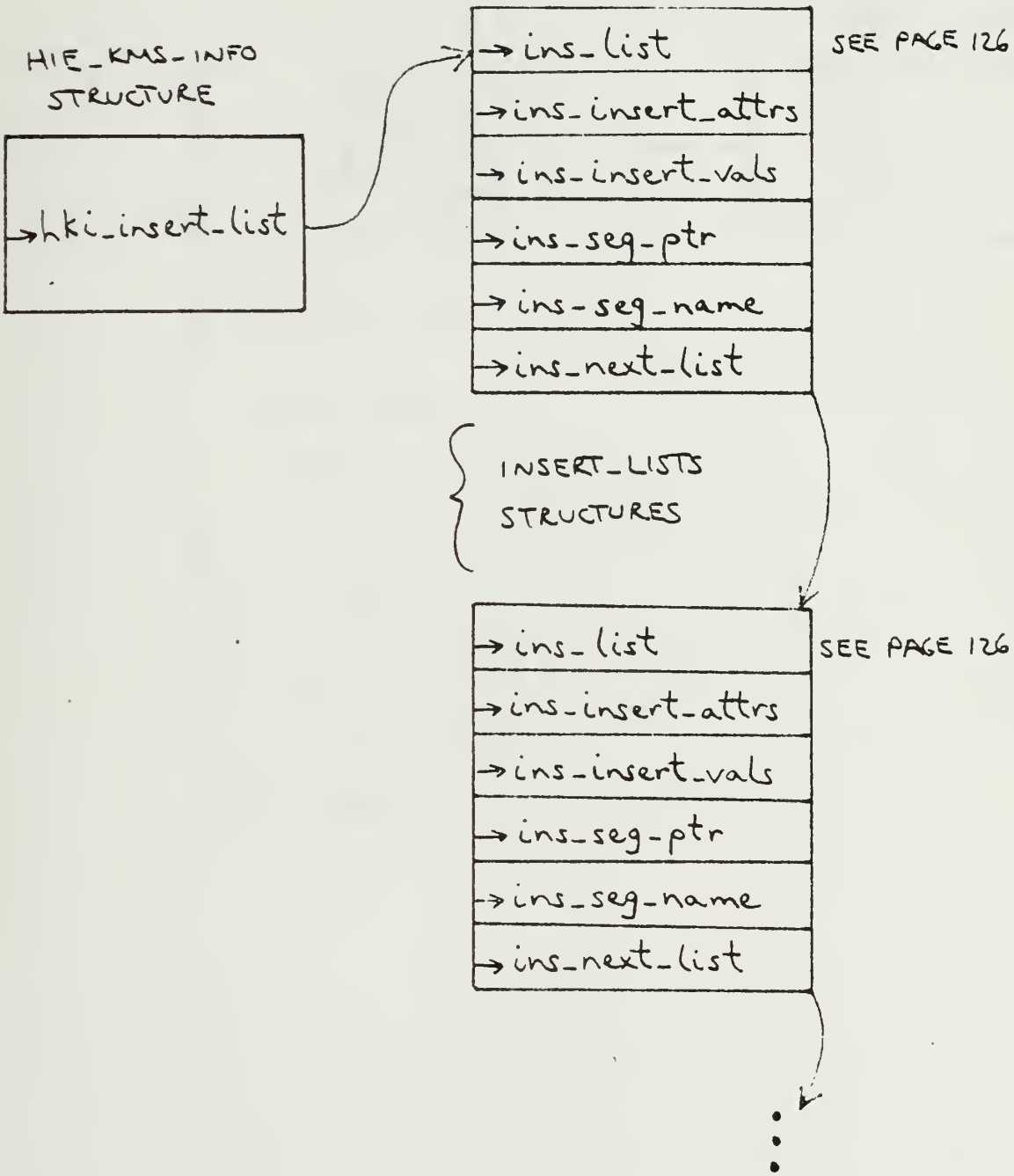


Figure 24. Continued

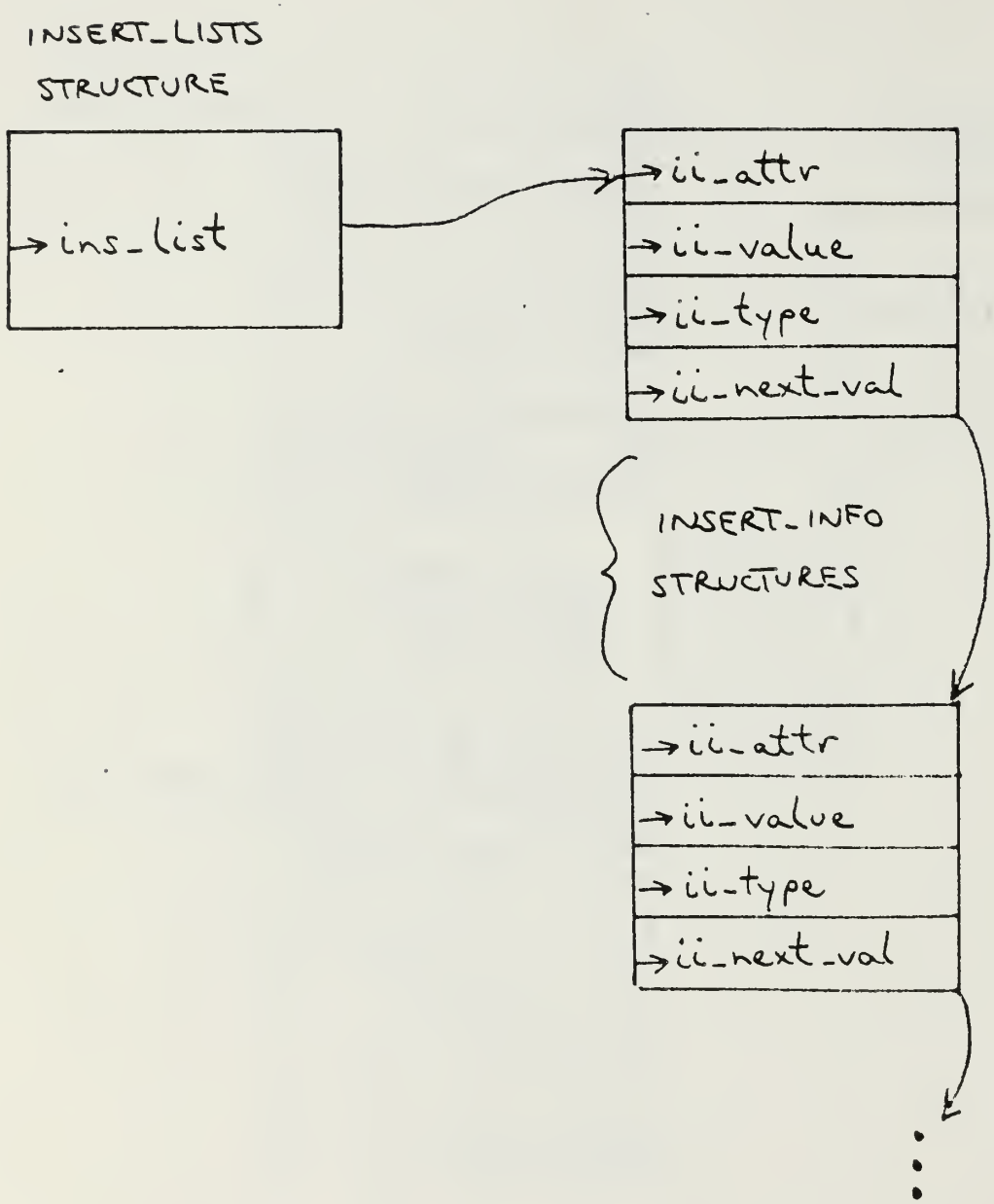


Figure 24. Continued

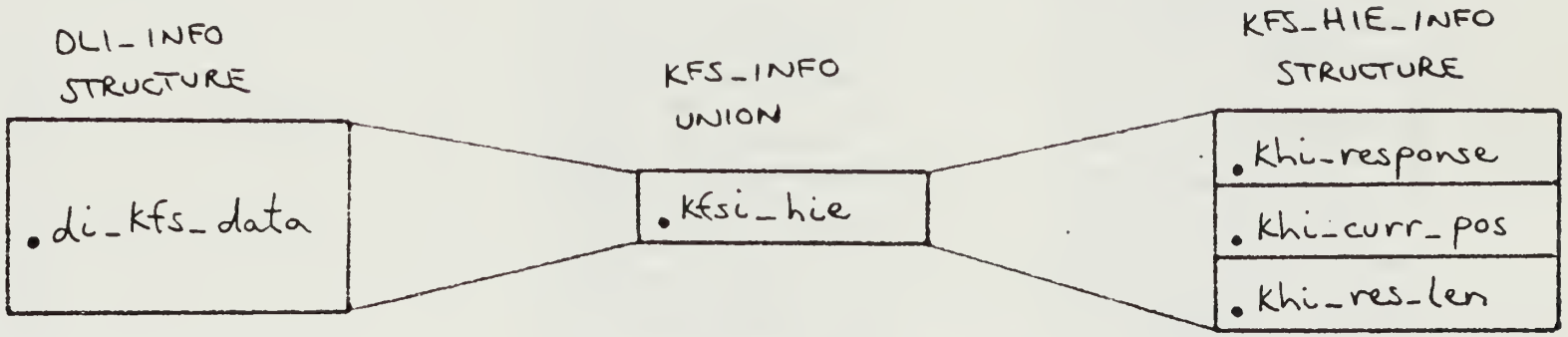


Figure 24. Continued

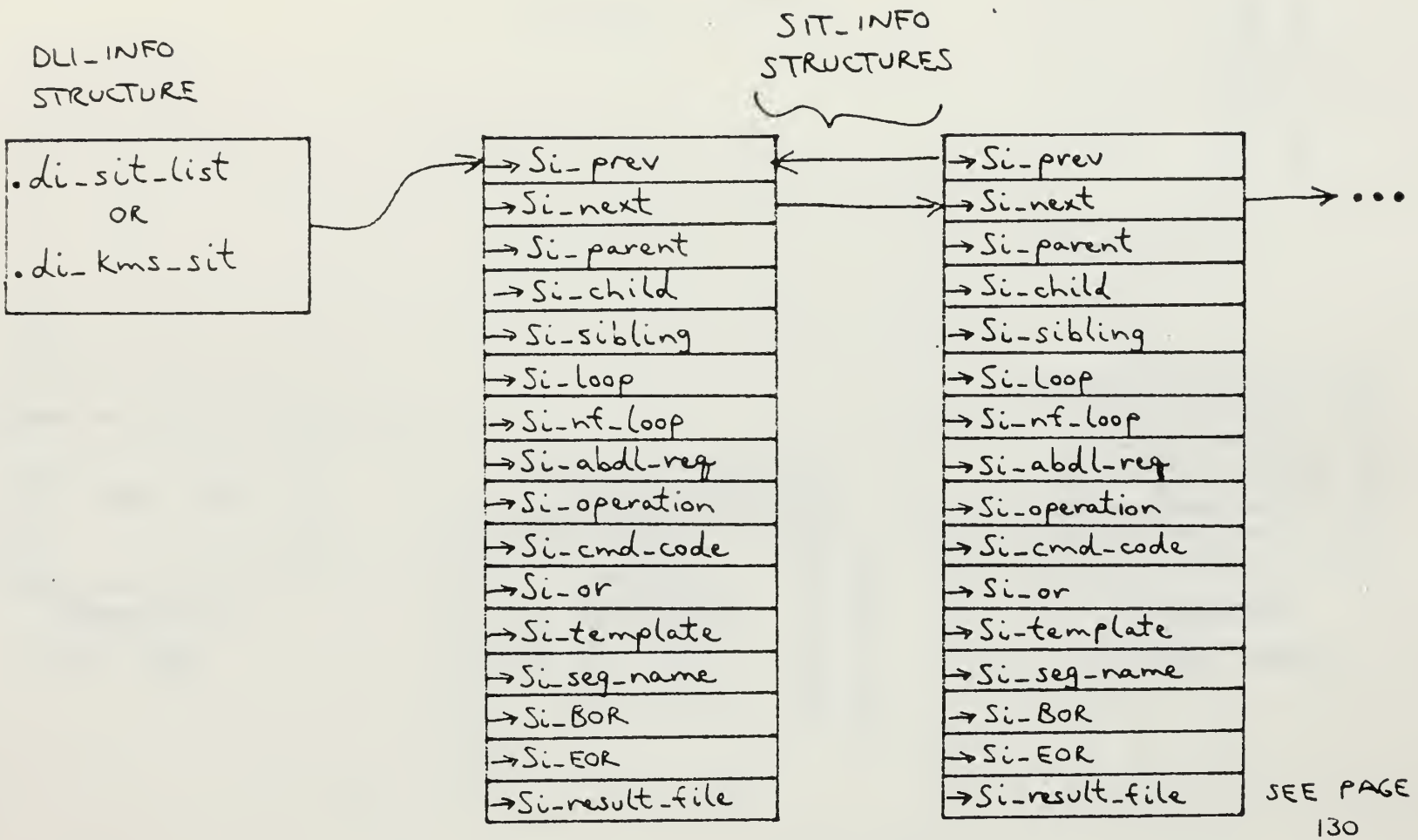


Figure 24. Continued

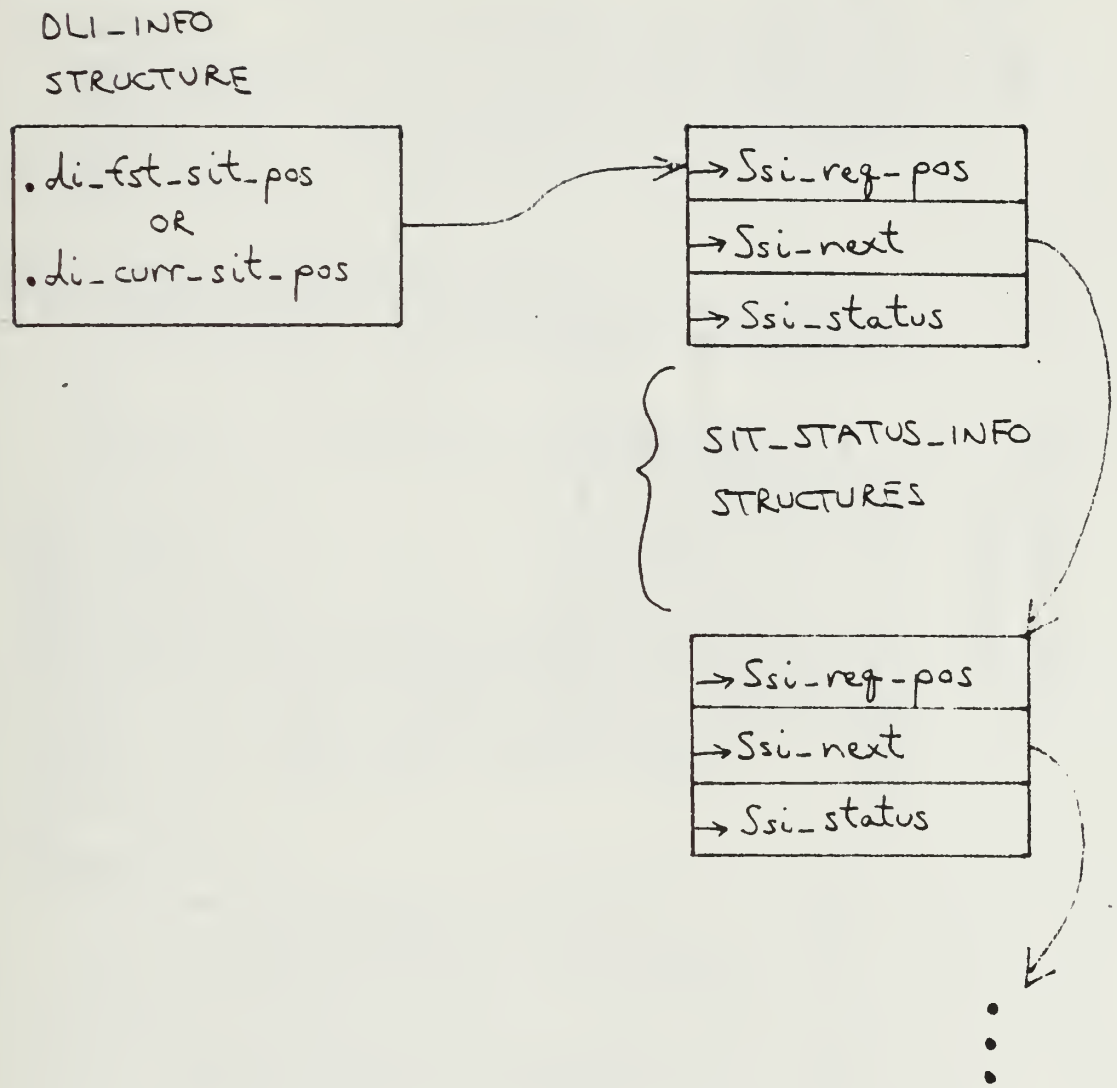


Figure 24. Continued

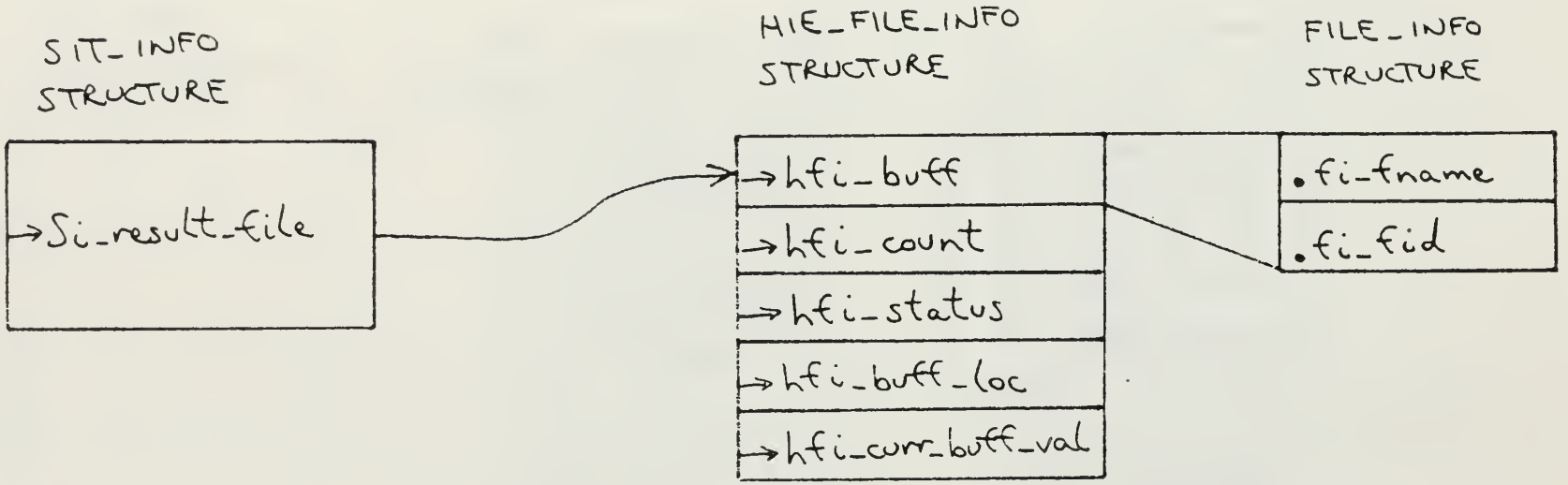


Figure 24. Continued

```
module DLI-INTERFACE

    db-list : list;        /* list of existing relational schemas       */
    head-db-list-ptr: ptr; /* ptr to head of the relational schema list */
    current-ptr: ptr;      /* ptr to the current db schema in the list  */
    follow-ptr: ptr;       /* ptr to the previous db schema in the list */
    db-id : string;        /* string that identifies current db in use  */


proc LANGUAGE-INTERFACE-LAYER();
    /* This proc allows the user to interface with the system.  */
    /* Input and output: user DLI requests                      */

      stop : int;  /* boolean flag */
      answer: char:  /* user answers to terminal prompts */

 perform DLI-INIT():
 stop = 'false';
 while (not stop) do
  /* allow user choice of several processing operations */
  print ("Enter type of operation desired");
  print ("    (l) - load new database");
  print ("    (p) - process existing database");
  print ("    (x) - return to the to operating system");
  read (answer);

  case (answer) of
    'l': /* user desires to load a new database */
         perform LOAD-NEW();
    'p': /* user desires to process an existing database */
         perform PROCESS-OLD();
    'x': /* user desires to exit to the operating system */
         /* database list must be saved back to a file   */
         store-free-db-list(head-db-list, db-list);
         stop = 'true';
         exit();
    default: /* user did not select a valid choice from the menu */
           print ("Error - invalid operation selected");
           print ("Please pick again")'
  end-case;

  /* return to main menu */
 end-while;

end-proc;
```

```
proc  DLI-INIT();

end-proc;


proc  LOAD-NEW();
     /*  This proc accomplishes the following:                    */
     /*  (1) determines if the new database name already exists,    */
     /*  (2) adds a new header node to the list of schemas,         */
     /*  (3) determines the user input mode (file/terminal),        */
     /*  (4) reads the user input and forwards it to the parser, and  */
     /*  (5) calls the routine that builds the template/descriptor files */

     answer: int;        /* user answer to terminal prompts */
     more-input: int;    /* boolean flag */
     proceed: int:       /* boolean flag */
     stop : int;         /* boolean flag */
     db-list-ptr: ptr;   /* pointer to the current database */
     req-str: str;       /* single create in DLI form */
     ptr-abdl-list: ptr: /* ptr to a list of ABDL queries (nil for this proc)*/
     tfid, dfid: ptr;    /* pointers to the template and descriptor files */


  /* prompt user for name of new database */
  print ("Enter name of database");
  readstr (db-id);
  db-list-ptr = head-db-list-ptr;

  stop = 'false';
  while (not stop) do
    /* determine if new database name already exists */
    /* by traversing list of relational db schemas   */
    if (db-list-ptr.db-id = existing db) then
      print ("Error - db name already exists");
      print ("Please reenter db name");
      readstr (db-id);
      db-list-ptr = head-db-list-ptr;
    end-if;
    else
      if (db-list-ptr + 1 = 'nil') then
        stop = 'true';
      else
        /* increment to next database */
        db-list-ptr = db-list-ptr + 1;
    end-else;

  end-while;
```

```
  /* continue - user input a valid 'new' database name */
  /* add new header node to the list of schemas and fill-in db name */
  /* append new header node to db-list */
  create-new-db(db-id);

  /* the KMS takes the DLI defines and builds a new list of relations   */
  /* for the new database. After all of the defines have been processed */
  /* the template and descriptor files are constructed by traversing    */
  /* the new database definition (schema).                              */

  more-input = 'true';
  while (more-input) do
    /* determine user's mode of input */
    print ("Enter mode of input desired");
    print ("    (f) - read in a group of defines from a file");
    print ("    (x) - return to the main menu");
    read (answer);

    case (answer) of
      'f': /* user input is from a file */
           perform READ-TRANSACTION-FILE();
           perform DBD-TO-KMS();
           perform FREE-REQUESTS();
           perform BUILD-DDL-FILES();
           perform KERNEL-CONTROLLER();

      'x': /* exit back to LIL */
           more-input = 'false';

      default: /* user did not select a valid choice from the menu */
           print ("Error - invalid input mode selected");
           print ("Please pick again");
    end-case;
  end-while;

end proc;
```

```
proc  PROCESS-OLD():
    /*  This proc accomplishes the following:                */
    /*  (1) determines if the database name already exists.  */
    /*  (2) determines the user input mode (file/terminal),  */
    /*  (3) reads the user input and forwards it to the parser */

    answer: int;        /* user answer to terminal prompts */
    found: int;         /* boolean flag to determine if db name is found */
    more-input: int;    /* boolean flag to return user to LIL */
    proceed: int;       /* boolean flag to return user to mode menu */
    db-list-ptr: ptr;   /* pointer to the current database */
    req-str: str;       /* single query in DLI form */
    ptr-abdl-list: ptr; /* pointer to a list of queries in ABDL form */
    tfid, dfid: ptr;    /* pointers to the template and descriptor files */


/* prompt user for name of existing database */
print ("Enter name of database");
readstr (db-id);
db-list-ptr = head-db-list-ptr;

found = 'false';
while (not found) do
  /* determine if database name does exist    */
  /* by traversing list of hierarchical schemas */
  if (db-id = existing db) then
    found = 'true';
  end-if;
  else
    db-list-ptr = db-list-ptr + 1;
    /* error condition causes end of list('nil') to be reached */
    if (db-list-ptr = 'nil') then
      print ("Error - db name does not exist");
      print ("Please reenter valid db name");
      readstr (db-id);
      db-list-ptr = head-db-list-ptr;
    end-if;

  end-else;

end-while;
```

```
  /* continue - user input a valid existing database name */
  /* determine user's mode of input */

  more-input = 'true';
  while (more-input) do
    print ("Enter mode of input desired");
    print ("    (f) - read in a group of DL/I requests from a file");
    print ("    (t) - read in a single DL/I request from the terminal");
    print ("    (x) - return to the previous menu");
    read (answer):

    case (answer) of
      'f': /* user input is from a file */
           perform READ-TRANSACTION-FILE();
           perform DLIREQS-TO-KMS();
           perform FREE-REQUESTS();

      't': /* user input is from the terminal */
           perform READ-TERMINAL();
           perform DLIREQS-TO-KMS();
           perform FREE-REQUESTS();

      'x': /* user wishes to return to LIL menu */
           more-input = 'false';

      default: /* user did not select a valid choice from the menu */
           print ("Error - invalid input mode selected");
           print ("Please pick again");
    end-case;

  end-while;

end-proc;


proc  READ-TRANSACTION-FILE();
  /* This routine opens a dbd/request file and reads the transactions */
  /* into the transaction list.  If open file fails, loop until valid */
  /* file entered                                                     */

  while (not open file) do
    print ("Filename does not exist");
    print ("Please reenter a valid filename");
    readstr ( file):
  end-while:

  READ-FILE();

end-proc;
```

```
proc READ-FILE();
    /* This routine reads transactions from either a file or the */
    /* terminal into the user's request list structure so that   */
    /* each request may be sent to the KERNEL-MAPPING-SYSTEM.    */

end-proc;


proc  READ-TERMINAL();
    /* This routine substitutes the STDIN filename for the read  */
    /* command so that input may be intercepted from the terminal */

end-proc;


proc DBD-TO-KMS();
    /* This routine sends the request list of database descriptions */
    /* one by one to the KERNAL-MAPPING-SYSTEM                      */

  while (more-dbds) do
    KERNAL-MAPPING-SYSTEM();
  end-while;

end-proc;
```

```
proc DLIREQS-TO-KMS();
    /* This routine causes the DL/I requests to be listed to the screen. */
    /* The selection menu is then displayed allowing any of the        */
    /* DL/I requests to be executed.                                   */

  perform LIST-DLIREQS();
  proceed = 'true';
  while (proceed) do
   print ("Pick the number or letter of the action desired");
   print ("    (num) - execute one of the preceding DL/I requests");
   print ("    (d)   - redisplay the file of DL/I requests");
   print ("    (r)   - reset currency pointer to the root");
   print ("    (x)   - return to the previous menu");
   read (answer);

   case (answer) of
    'num' : /* execute one of the requests */
          traverse query list to correct query;
          perform KERNAL-MAPPING-SYSTEM();
          perform KERNEL-CONTROLLER();

    'd'   : /* redisplay requests */
          perform LIST-DLIREQS();

    'r'   : /* reset currency ptr to the root */
          perform CURR-PTR-TO-ROOT();

    'x'   : /* exit to mode menu */
          proceed = 'false';

    default : /* user did not select a valid choice from the menu */
          print (" Error - invalid option selected");
          print (" Please pick again");
   end-case;

  end-while;

end-proc;
```

```
proc  kernel-mapping-system ()
  perform  parser()
  perform  match()
end-proc  kernel-mapping-system



proc  parser()
  if  (operation != CreateDB, vice work with existing DB)
    alloc and init initial kms data structures
    access and save length of dli request .
    free any existing abdl-str(s) from a previous parse
  end-if

  initialize the input request ptr
  perform  yyparse()
  reset all booleans and. counter variables

  if  (operation != CreateDB)
    free all kms-unique data structures
  end-if
end-proc  parser


proc  yyparse ()

     /*  This procedure accomplishes the following :                  */
     /*  (1) parses the DLI input requests and maps them to appropriate  */
     /*      abdl requests, using LEX and YACC to build proc yyparse().  */
     /*  (2) builds the hierarchical schema, when loading a new db.      */
     /*  (3) checks for validity of segment and attribute names within    */
     /*      the given db schema, when processing requests against an     */
     /*      existing db.                                                 */
```

```
%{
    boolean: creating       /* signals a DBLoad vs a DBQuery */
    boolean: updating       /* signals a DLI update request */
    boolean: label          /* signals DLI statement has a label */
    boolean: or-where       /* signals an OR term in SSA predicate */
    boolean: and-where      /* signals an AND term in SSA predicate */
    boolean: literal-const  /* signals alpha constant vs integer constant */
    boolean: inserting      /* signals ISRT operation */
    boolean: not-marked     /* label not marked for attachment of loop ptr*/
    boolean: first-ssa      /* signals working on 1st ssa of DLI request */
    boolean: seq-fld-has-value  /* the 'value' of seq-fld is given in req */
    boolean: missing-root   /* missing root seg in ssa specif of DLI req */
    boolean: goto-found     /* GOTO found following last ssa in DLI req */
    boolean: spec-ret-op    /* special retrieve op (GN or GNP -- all segs) */
    boolean: single-build   /* single segment to be built for ISRT op */
    boolean: star-d         /* cmd code D used in DLI ISRT op */
    ptr: seg-ptr            /* ptr to a schema segment */
    ptr: curr-seg-ptr       /* ptr to current segment in schema */
    ptr: prev-seg-ptr       /* ptr to previous segment in schema */
    ptr: parent-ptr         /* ptr to parent of current segment, in schema */
    ptr: curr-1st-child-ptr /* ptr to 1st child of curr parent in schema */
    ptr: label-ptr          /* ptr to abdl-str that corresponds to label */
    int: attr-len           /* length of current attribute */
    int: insert-attrs       /* number of attrs inserted during ISRT op */
    int: insert-vals        /* number of vals inserted during ISRT op */
    int: operator-flag      /* dli-operator in DLI request */
    int: addl-tgt-count     /* count of add'l items added to tgt-list */
    int: build-count        /* count of number of segments built for ISRT op */
    int: ssa-count          /* count of the number of ssa's in multiple ISRT */

    char: cmd-code          /* command codes 'D', 'F', or 'V' */
    char: attr-type         /* 's'=CHAR, 'i'=INT, 'f'=FLOAT, from schema */
    char: data-type         /* same as attr-type, for an input 'value' */
    str: label-name
    str: segment-name
    str: field-name
    str: abdl-str
    str: temp-str
    flag: loop-flag         /* there's a GOTO 'label' loop in curr request */
    list: tgt-list          /* list of sequence field attribute names */
    list: insert-list       /* list of attribute-value pairs for ISRT op */
    list: insert-nodes      /* list of insert-list(s) for multiple ISRT op */

%}
```

```
%token    /* List All Tokens From "LEX", and their TYPE, here */

%start  statement


%%


statement:  dml-statement
            {
            if  (! spec-ret-op)
              save last curr-seg-ptr
            end-if
            return
            }
          | ddl-statement
            {
            return
            }
            ;

ddl-statement:  db-desc  segment-list
                ;
db-desc:  DBD
          {
          creating = TRUE
          curr-1st-child-ptr = NULL
          }
         NAME  EQ  db-name
          {
          locate dbid schema header node
          if  (db names do not match)
            print ("Error - given db-name doesn't match db-name in file")
            perform yyerror()
            return
          end-if
          }
          ;

segment-list:  segment-desc
             | segment-list  segment-desc
               ;

segment-desc:  segment  field-list
               ;
```

```
segment: SEGM NAME EQ
        {
        allocate and init a new segment-node
        dbid-node num-seg++
        }
       segment-spec
        :

segment-spec: segment-name
            {
            if (! valid-child(segment-name, curr-1st-child-ptr) )
              copy segment-name to current segment-node
            end-if
            else
              print ("Error - 'segment-name' segment doubly defined in db")
              perform yyerror()
              return
            end-else
            }
           A
            ;
```

```
A:  empty
      {
      connect new segment-node to the dbid root-ptr
      }
    | COMMA  PARENT  EQ  segment-name
      {
      seg-ptr = the root segment of the db
      if  ( valid-parent(seg-ptr, segment-name, parent-ptr) )
        connect the new segment-node to the appropriate parent-node
        establish curr-1st-child-ptr
        parent-node num-child++
        first-child and sibling node(s) num-sib++
      end-if
      else
        print ("Error - 'segment-name' parent-node does not exist")
        perform yyerror()
        return
      end-else
      }
      ;

field-list:  field-desc
             {
             connect new attr-node to segment-node
             }
           | field-list  field-desc
             {
             connect successive attr-node(s) to segment-node
             }
             ;

field-desc:  FIELD  NAME  EQ
             {
             allocate and init a new attr-node
             segment-node num-attr++
             }
            field-spec
             ;
```

```
field-spec: field-name
            {
            if ( valid-attribute(seg-ptr, field-name, &attr-len, &attr-type) )
              print ("Error - 'field-name' attr doubly defined in 'segment-name'")
              perform yyerror()
              return
            end-if
            else
              copy field-name to attr-node
            end-else
            attr-node key-flag = 0
            attr-node multiple field = 0
            }
          COMMA field-data
          LPAR field-name
            {
            if ( valid-attribute(seg-ptr, field-name, &attr-len, attr-type) )
              print ("Error - 'field-name' attr doubly defined in 'segment-name'")
              perform yyerror()
              return
            end-if
            else
              copy field-name to attr-node
            end-else
            }
          COMMA SEQ B RPAR COMMA field-data
            {
            attr-node key-flag = 1
            }
            ;

B: empty
    {
    attr-node multiple field = 0
    }
  COMMA M
    {
    attr-node multiple field = 1
    }
    ;

field-data: C BYTES EQ INTEGER
            {
            attr-node length = INTEGER
            }
            ;
```

```
C: empty
    {
    attr-node type = 's'        /* default condition */
    }
  | TYPE EQ data-type COMMA
    ;

data-type: CHAR
           {
           attr-node type = 's'
           }
         | INT
           {
           attr-node type = 'i'
           }
         | FLT
           {
           attr-node type = 'f'
           }
           ;

dml-statement: J ssa
               {
               if ( (label) and (! goto-found) )
                 print ("Warning - 'label-name' label defined, but not referenced")
                 perform yyerror()
                 return
               end-if
               if (operator-flag = ISRT)
                 if ( (single-build) and (star-d) )
                   print ("Error - '*D' cmd code implies a multiple seg ISRT")
                   perform yyerror()
                   return
                 end-if
                 if ( (! single-build) and (! star-d) )
                   print ("Error - '*D' cmd code req'd for mutiple seg ISRT")
                   perform yyerror()
                   return
                 end-if
                 if (! single-build)
                   if (build-count != ssa-count)
                     print ("Error - num of segs built not equal to num ssa's")
```

```
    perform yyerror()
    return
   end-if
  end-if
  for  (each node in the list of insert-lists)
   if  (! single-build)
    re-establish the saved curr-seg-ptr and segment-name
   end-if
   if  (insert-attrs < 1)
    copy all segment attrs to insert-list and count insert-attrs
   end-if
   if  (insert-attrs != insert-vals)
    print ("Error - too many, or not enough values inserted")
    perform yyerror()
    return
   end-if
   for  (each attribute in the curr-seg)
    if  (segment attribute missing from insert list)
     add item to insert-list with default values
     of 'Zz' if type CHAR, and '0' if type INT
    end-if
    else
     if  (insert-list item = seq-fld of curr-seg)
      seq-fld-has-value = TRUE
     if  (! seq-fld-has-value)
      print ("Error - seq-fld value req'd in ISRT op")
      perform yyerror()
      return
     end-if
     if  (! valid-attribute(curr-seg-ptr, field-name, &attr-len, &attr-type) )
   print ("Error - 'field-name' attr does not exist in 'segment-name' segment")
      perform yyerror()
      return
     end-if
     if  (insert-list data-type != attr-type)
      print ("Error - 'field-name' attr must be type 'attr-type'")
      perform yyerror()
      return
     end-if
     if  (attr-len < strlen(insert-list value))
      print ("Error - 'field-name' attr max length = 'attr-len'")
      perform yyerror()
      return
     end-if
    end-else
   end-for
```

```
    if ( (seq-fld-has-value) and (single-build) )
      delete last abdl-str present
      (ie, seq-fld given, no retrieve request required)
    end-if
    alloc and init a new abdl-str
    copy "[ INSERT (<TEMPLATE, 'segment-name'>" to abdl-str
    if (single-build)
      for (each item in tgt-list)
        concat ", <'seq-fld', ***...***>" to abdl-str
      end-for
    end-if
    else
      for (each node in the list of insert-lists)
        concat ", <'first attr-name', 'first attr-value'>" to abdl-str
      end-for
    end-else
    for (each item in insert-list)
      concat ", <'attr-name', 'attr-value'>" to abdl-str
    end-for
    concat ") ]" to abdl-str
  end-for
end-if

if ( (! spec-ret-op) and (single-build) )
  for (all abdl-str(s), except the last one)
    concat tgt-list and BY-clause to RETRIEVE reqs
    (ie, "('tgt-list') BY 'seq-fld' ]" )
    if (operator-flag = ISRT) and (cmd-code = StarD)
      retrieve all attr names and add to tgt-list
    end-if
  end-for

  concat all attrs to last RETRIEVE request
  concat ") BY 'sequence-field' ]" to last RETRIEVE request

  if (operator-flag = DLET)
    form the descendant deletes to complete the DLET req
  end-if
end-if

if (spec-ret-op)
  if (operator-flag = GN)
    form the descendant retrieves to complete the GN (no ssa) req
  end-if
  else if (operator-flag = GNP)
    form retrieves for children to complete the GNP (no ssa) req
  end-else-if
```

```
            else
              print ("Error - seg-name must be specified if GN or GNP op")
              perform yyerror()
              return
            end-else
          end-if
          }
          ;

J: empty
 | H
    {
    label = TRUE
    save label-name ('H') for later comparison with GOTO statement
    }
   -;

dli-operator: empty | GU | GN | GNP | GHU | GHN | GHNP | build-segs ISRT
          {
          set appropriate operator-flag
          }
          ;

build-segs: build-segment
          {
          build-count = build-count + 1
          }
        | build-segs build-segment
          {
          build-count = build-count + 1
          single-build = FALSE
          }
          ;

build-segment: BUILD I COLON
           {
           inserting = TRUE
           }
         value-list
           :

I: empty
    {
    alloc and init insert-list node
    }
 | LPAR field-name-list RPAR
    ;
```

```
field-name-list:  field-name
                  {
                  alloc and init insert-list node
                  alloc first insert-list item
                  copy 'field-name' to insert-list
                  insert-attrs ++
                  }
                | field-name-list  COMMA  field-name
                  {
                  alloc next insert-list item
                  copy 'field-name' to insert-list
                  insert-attrs ++
                  }
                  ;

value-list:  LPAR  constant-list  RPAR
             {
             inserting = FALSE
             if  (insert-attrs > 0)
               if  (insert-attrs != insert-vals)
                 print ("Error - too many, or not enough values inserted")
                 perform yyerror()
                 return
               end-if
             end-if
             }
             ;

constant-list:  constant
                {
                if  (insert-attrs < 1)
                  alloc first insert-list item
                end-if
                if  (literal-const)
                  convert-AlphaNumFirst('constant')
                  literal-const = FALSE
                  copy data-type = 's' to insert-list
                end-if
                else
                  copy data-type = 'i' to insert-list
                end-else
                copy 'constant' to insert-list
                insert-vals ++
                }
```

```
        constant-list  COMMA  constant
          {
          if  (insert-attrs < 1)
            alloc next insert-list item
          end-if
          if  (literal-const)
            convert-AlphaNumFirst('constant')
            literal-const = FALSE
            copy data-type = 's' to insert-list
          end-if
          else
            copy data-type = 'i' to insert-list
          end-else
          copy 'constant' to insert-list
          insert-vals ++
          }
          ;

E:  empty
    {
    if  (label)
      print ("Warning - 'label-name' label defined, but not referenced")
    end-if
    }
  GOTO  H
    {
    goto-found = TRUE
    if  ( (! label)  or  ( (label)  and  ('H' != 'label-name') ) )
      print ("Error - label for 'GOTO H' not defined")
      perform yyerror()
      return
    end-if
    if  (op-flag != GnOp, or GnpOp, or IsrtOp)
      print ("Error - loops used only w/ GN, GNP, or ISRT operations")
      perform yyerror()
      return
    end-if
    if  ( (! spec-ret-op) and (single-build) )
      set loop-flag for use by KC
    end-if
    else if  (! single-build)
      print ("Error - loops cannot be used w/ multiple ISRT ops")
      perform yyerror()
      return
    end-else-if
    }
```

```
| NFGOTO H
    {
    goto-found = TRUE
    if ( (! label) or ( (label) and ('H' != 'label-name') ) )
      print ("Error - label for 'NFGOTO H' not defined")
      perform yyerror()
      return
    end-if
    if (op-flag != GnOp, or GnpOp, or IsrtOp)
      print ("Error - loops used only w/ GN, GNP, or ISRT operations")
      perform yyerror()
      return
    end-if
    if ( (! spec-ret-op) and (single-build) )
      set loop-flag for use by KC
    end-if
    else if (! single-build)
      print ("Error - loops cannot be used w/ multiple ISRT ops")
      perform yyerror()
      return
    end-else-if
    }
    ;

K: empty
 | dli-op
    ;

dli-op: DLET
        {
        if (not preceeded by a GET HOLD operation)
          print ("Error - DLET must be preceeded by GHU, GHN, or GHNP")
          perform yyerror()
          return
        end-if
        else
          op-flag = DLET
          alloc and init a new abdl-str
          /* formulate the first DELETE request */
          copy "[ DELETE ((TEMPLATE = 'segment-name')" to abdl-str
          for (ea item in the tgt-list)
            concat " and ('item-name' = ***...***)" to abdl-str
          end-for
          concat ") ]" to abdl-str
        end-else
        }
```

```
        chg-pred  REPL
          {
          if  (not preceeded by a GET HOLD operation)
            print ("Error - REPL must be preceeded by GHU, GHN, or GHNP")
            perform yyerror()
            return
          end-if
          else
            op-flag = REPL
          end-else
          }
          ;

chg-pred:  CHANGE
          {
          updating = TRUE
          }
          field-name  TO  constant
          {
          updating = FALSE
          if  (! valid-attribute(curr-seg-ptr, field-name, &attr-len, &attr-type) )
            print ("Error - 'field-name' attr does not exist in 'segment-name' segment")
            perform yyerror()
            return
          end-if
          if  (literal-const)
            convert-AlphaNumFirst('constant')
            literal-const = FALSE
            data-type = 's'
          end-if
          else
            data-type = 'i'
          end-else
          if  (data-type != attr-type)
            print ("Error - 'field-name' attr must be type 'attr-type'")
            perform yyerror()
            return
          end-if
          if  (attr-len < strlen('constant'))
            print ("Error - 'field-name' attr max length = 'attr-len'")
            perform yyerror()
            return
          end-if
```

```
        alloc and init a new abdl-str
        copy "[ UPDATE ((TEMPLATE = 'segment-name')" to abdl-str
        for (each item in tgt-list)
          concat " and ('seq-fld' = ***...***)" to abdl-str
        end-for
        concat ") <'field-name' = 'constant'> ]" to abdl-str
        }
        ;

ssa: seg-srch-arg
      {
      ssa-count = ssa-count + 1
      prev-seg-ptr = curr-seg-ptr
      if ( (operator-flag = ISRT) and (! single-build) )
        save the curr-seg-ptr and segment-name in first insert-list node
      end-if
      first-ssa = FALSE
      }
   | ssa seg-srch-arg
      {
      ssa-count = ssa-count + 1
      if (the parent of the curr-seg-ptr = prev-seg-ptr)
        prev-seg-ptr = curr-seg-ptr
      end-if
      else
        print ("Error - SSA specifies incorrect hierarchical path")
        perform yyerror()
        return
      end-else
      if ( (operator-flag = ISRT) and (! single-build) )
        save the curr-seg-ptr and segment-name in first insert-list node
      end-if
      }
      ;

seg-srch-arg: dli-operator segment-name
            {
            seg-ptr = the root segment of the db
            if (! valid-parent(seg-ptr, 'segment-name', curr-seg-ptr) )
              print ("Error - 'segment-name' segment does not exist")
              perform yyerror()
              return
            end-if
            if ( (operator-flag != ISRT) or (single-build) )
              alloc and init a new abdl-str and a new tgt-list item
              copy "[ RETRIEVE (" to abdl-str
              copy segment sequence field and length to tgt-list
            end-if
```

```
            if  ( (label) and (not-marked) )
              not-marked = FALSE
              label-ptr = current abdl-str
            end-if
            if  ( (first-ssa) or (missing-root) )
              if  (curr-seg-ptr = root of the db)
                insert seq-fld(s) to tgt-list for all parents/grandparents
                addl-tgt-count = number inserted
                missing-root = TRUE
              end-if
            end-if
            save 'segment-name' for later use
            }
          L  G
            {
            delete first 'addl-tgt-count' items from tgt-list
            addl-tgt-count = 0
            if  (single-build)
              concat ") " to abdl-str
            end-if
          E  K
            }
        | dli-operator
            {
            spec-ret-op = TRUE;
            }
          E  K
            ;

L:  empty
  | ASTERISK  N
    ;

N:  D  |  F  |  V
    {
    set cmd-code to appropriate code (StarF, or StarV)
    if  (N is D)
      star-d = TRUE
      if  (single-build)
        set cmd-code to StarD
      end-if
    end-if
    }
    ;
```

```
G: empty
    {
    if (! single-build)
      do nothing
    end-if
    else
      if (curr-seg-ptr = root of the db)
        concat "TEMPLATE = 'segment-name'" to abdl-str
      end-if
      else
        concat "(TEMPLATE = 'segment-name')" to abdl-str
        for (ea item in tgt-list)
          concat " and ('item-name' = ***...***)" to abdl-str
        end-for
      end-else
    end-else
    }
  | LPAR boolean RPAR
    {
    if (or-where)
      concat ")" to abdl-str
      or-where = FALSE
    end-if
    }
    ;

boolean: boolean-term
         {
         concat "(TEMPLATE = 'segment-name') and " to abdl-str
         form symbolic id predicates : "('seq-fld' = ***...***)" from tgt-list,
           for all previous segments, and concat them to the abdl-str, each one
           separated by " and ".
         concat temp-str to abdl-str
         }
       | boolean OR
         {
         or-where = TRUE
         abdl-str[11] = '('
         concat ") or ((TEMPLATE = 'segment-name') and " to abdl-str
         copy 'empty str' to temp-str
         }
         boolean-term
         {
         form symbolic id predicates : "('seq-fld' = ***...***)" from tgt-list,
           for all previous segments, and concat them to the abdl-str, each one
           separated by " and ".
         concat temp-str to abdl-str
         }
         ;
```

```
boolean-term:  boolean-factor
        |  boolean-term  AND
            {
            and-where = TRUE
            concat " and " to temp-str
            }
           boolean-factor
            ;

boolean-factor:  predicate
              ;

predicate:  field-name
           {
           if  (! valid-attribute(curr-seg-ptr, field-name, &attr-len, &attr-type) )
             print ("Error - 'field-name' attr does not exist in 'segment-name' segment")
             perform yyerror()
             return
           end-if
           else
            if  ( (! and-where) and (! or-where) )
              alloc temp-str
              copy "(" to temp-str
            end-if
            else
              concat "(" to temp-str
            end-else
            concat 'field-name' to temp-str
            save 'field-name' for later use
            and-where = FALSE
           end-else
           }
         comparison
           {
           concat " 'comparison' " to temp-str
           }
         constant
           {
           if  (literal-const)
             convert-AlphaNumFirst('constant')
             literal-const = FALSE
           end-if
           concat "'constant')" to temp-str
           }
           ;
```

```
comparison: EQ | NE | LT | LE | GT | GE
          ;

constant: QUOTE H QUOTE
          {
          literal-const = TRUE
          if ( (! inserting) and (! updating) )
           if (attr-type != 's')
            print ("Error - 'field-name' attr must be type INT")
            perform yyerror()
            return
           end-if
           if (attr-len < strlen('H'))
            print ("Error - 'field-name' attr max length = 'attr-len'")
            perform yyerror()
            return
           end-if
          end-if
          }
        | INTEGER
          {
          if ( (! inserting) and (! updating) )
           if (attr-type != 'i')
            print ("Error - 'field-name' attr must be type CHAR")
            perform yyerror()
            return
           end-if
           if (attr-len < strlen('INTEGER'))
            print ("Error - 'field-name' attr max length = 'attr-len'")
            perform yyerror()
            return
           end-if
          end-if
          }
          ;

H: IDENTIFIER
 | VALUE
   ;

db-name: IDENTIFIER
       ;

segment-name: IDENTIFIER
            ;

field-name: IDENTIFIER
          ;
```

```
empty: ;

end-proc yyparse


%%



proc yyerror(s)
      char s
 if (operation = CreateDB)
   set error flag for the LIL
   print ("Error - DBD Description file for 'curr-seg' in error")
   free all the malloc'd variables in the current schema
 end-if
 else
   set error flag for the LIL
   free all the malloc'd variables in the kms data structures
 end-else

 reset all boolean and counter variables
 print (s)
end-proc yyerror
```

```
proc MATCH()
      /* This routine checks the operator flag for the incoming     */
      /* transaction and branches to the appropriate DL/I operation */

      kms-list : list;
      sit-list : list;
      status-list : list;
      head-kms-list-ptr : ptr;
      head-sit-list-ptr : ptr;
      status-ptr : ptr;
      sit-ptr : ptr;
      first-status-node-ptr : ptr;
      curr-status-node-ptr : ptr;


  /* the kms list cannot be null */
  if (kms-list <> 'null')
    case (kms-list.operation)
      "GhuOp" : /* Get hold unique operator */
                perform GET-HOLD-UNIQUE();

      "IsrtOp" : /* Insert operator */
                perform INSERT();

      "GuOp" : /* Get Unique operator */
                perform GET-UNIQUE();

      "GnOp" : /* Get Next operator */
                perform GET-NEXT();

      "SpecRetOp" : /* Special Retrieve operator */
                perform SPECIAL-RETRIEVE();

      "GnpOp" : /* Get Next Within Parent operator */
                perform GET-NEXT-PARENT();
    end-case;

end-proc;
```

```
proc GET-HOLD-UNIQUE()
      /* A GHU opration allows one user exclusive access to the database  */
      /* so that subsequent deletes or replaces will occur before any     */
      /* further users can access the database.                           */

      dlet-flag : int;  /* boolean flag to tell if found a DELETE op */
      done : int;       /* boolean flag                              */

 if (sit-list <> 'null')
   print ("Error - sit-list is not null as required for GhuOp");
 else
     /* When a GHU is found, the type of operation must be identified.   */
     /* The kms list is scanned looking for a delete operator. If found, */
     /* a status node is created and set to point to the first node of   */
     /* the kms list (the GHU). A second status node is created and set  */
     /* to the kms node that has the beginning-of-request delete flag    */
     /* set. If the delete operator is not found, this indicates a       */
     /* replace operation or a list with nothing but GHU's. In this case */
     /* a status node is created and set to point to the first node of   */
     /* the kms list (the GHU).                                          */

     done = 'false';
     dlet-flag = 'false';
     sit-ptr = kms-list + 1;

     /* walk down sit list until find DELETE operator or empty list */
     while (sit-ptr <> 'null' OR not done)
       if (sit-ptr.operation = DletOp)
         /* case of DELETE operation */
         if (fisrst-status-list-ptr = 'null')
           /* case of status list being empty */
           allocate a new status node;
           first-status-list-ptr = new status node;
           curr-status-list-ptr = new status node;
           status-ptr = head-kms-list-ptr;
           allocate a new status node;
           append status node to the status list;
           status-ptr = sit-ptr;
         end-if;
         else
           print ("Error - status list not null as required for GhuOp");
         end-else:
         dlet-flag = 'true';
         done = 'true';
       end-if;
       sit-ptr = sit-ptr + 1;
     end-while;
```

```
    if (dlet-flag = 'false')
      /* case that no DELETE operators were found in sit list; this */
      /* indicates that we have REPLACE operations or just GHU'S.   */
      allocate a new status node;
      first-status-node-ptr = new status node;
      curr-status-node-ptr = new status node;
      status-ptr = head-kms-list-ptr;
    end-if:

    /* set sit list ptr to heading of the kms list and null out kms ptr */
    head-sit-list-ptr = head-kms-list;
    head-kms-list = 'null';
  end-else;

end proc;

proc INSERT()
      /* An insert operation is used to add a new segment, "node" */
      /* to the database.                                         */

      first-bor : int;   /* boolean flag set when beginning-of-req found */

  /* An insert operation can only access the database from the root. */
  /* As long as the sit list is null, then we know that the currency */
  /* pointer is pointing to the root.                                */
  if (sit-list <> 'null')
    printf("Error - sit list not null as required for IsrtOp");
  else
    /* set sit ptr to the head of the kms list */
    sit-ptr = head-kms-list-ptr;
    first-bor = TRUE;
```

```
  /* walk down the kms list until it is empty *
  while (sit-ptr <> 'null')
    /* When an insert is detected, the kms list is scanned and a */
    /* status node is created and set to point to each kms node  */
    /* that contains a beginning-of-request flag. The kms list   */
    /* is then transferred to the sit list before exiting.       */

    if (sit-ptr.BOR = 'true')
      allocate a new status node:
      if (first-bor = 'true')
        /* case of the status node being the first on the list */
        first-status-node-ptr = new status node;
        curr-status-node-ptr = new status node;
        first-bor = 'false':
      end-if;
    else
       append the status node onto status list;
      end-else;

      /* fill in the status node's contents */
      status-ptr = sit-ptr;
    end-if;

    sit-ptr = sit-ptr - 1;
  end-while;

  /* set sit list ptr to head of the kms list and null out kms ptr */
  head-sit-list-ptr = head-kms-list-ptr;
  head-kms-list-ptr = 'null';
 end-else;

end-proc;
```

```
proc GET-UNIQUE()
      /* A GU operation is used to access the database from the */
      /* root of the database. */


 /* A GU operation can only access the database from the root. */
 /* As long as the sit list is null, then we know that the    */
 /* currency pointer is pointing to the root.                 */
 if (sit-list <> 'null')
  print ("Error - sit-list not null as required for GuOp");
 else
  if (first-status-node-ptr = 'null')
    /* When a legitimate GU is found, we are sure that the  */
    /* the currency of the request is correct. In this case */
    /* we simply transfer the sit list for the GU from the  */
    /* kms list to the sit list and create a single status  */
    /* node that points to the first request of the GU.     */

    /* allocate a status node */
    allocate a new status node;

    /* set head of the status list to the allocated node */
    first-status-node-ptr = new status node;
    curr-status-node-ptr = new status node;

    /* fill in the contents of the allocated node */
    status-ptr = head-kms-list-ptr;

    /* set sit list ptr to heading of the kms list and null out kms ptr */
    head-sit-list-ptr = head-kms-list-ptr;
    head-kms-list-ptr = 'null';
  end-if;
  else
    print ("Error - status list not null as required for GuOp");
  end-else:

 end-else;

end-proc;
```

```
proc GET-NEXT()
      /* A GN operation is used to access the next lower level of the    */
      /* database.  It is used only after a GU operation has established */
      /* a currency ptr to a specific level of the database.             */

      found, done;      /* boolean flags */

      prev-kms-ptr;     /* ptr to the previous node on kms list */
      prev-sit-ptr;     /* ptr to the previous node on sit list */

 if (head-sit-list-ptr = 'null')
  /* with the sit list beint null, a GN is the same as a GU if the name */
  /* of first node of the kms list is the same as the root segment      */
  if (head-kms-list-ptr.seg-name = root segment name)
    perform GET-UNIQUE();
  end-if;
  else
    print ("Error - currency pointer must be set to the root");
    print (" or specify complete path");
  end-else;

 else
  if (first-status-node-ptr = 'null')
    print ("Error - status list null for GnOp");
  end-if;
  else
    /* When a valid GN is found, we know the segment that we want is    */
    /* the next occurrance of a legitimate child or the segment the     */
    /* currency pointer points to. If the segment is a child then we    */
    /* create a status node pointing to the first node of the kms list. */
    /* By being a child we guarantee ourselves that part of the kms     */
    /* list matches some of the sit list so the status field of the     */
    /* allocated status node is set to MATCHPART. If the segment is not */
    /* a child but the current segment, the amount of overlap between   */
    /* the sit list and the kms list must be checked. The parent pointer*/
    /* of the first kms node is set to the node above the node that it  */
    /* matches in the sit list. A status node is created pointing to the*/
    /* first node of the kms list. The kms and sit lists are then       */
    /* checked to see how much overlap they contain. The status field   */
    /* is set to MATCHPART or MATCHALL as appropriate.                  */

    dli-ptr->di-saved-seg-ptr2->hn-first-child->hn-name);
    if (head-kms-list-ptr.seg-name is a valid child of the node
                  currently pointed to by the currency pointer)
      /* segment we want the next of is a child of the current segment*/
      status-ptr = head-status-node-ptr;
```

```
    /* walk down to the end of the status list */
    while (status-ptr.next <> 'null')
      status-ptr = status-ptr + 1;
    allocate a new status node;
    status-ptr = head-kms-list;
    status-ptr.status = MATCHPART;
    append status node to the status list;

    /* walk down to the end of the sit list */
    sit-ptr = head-sit-list-ptr;
    while (sit-ptr.next <> 'null')
      sit-ptr = sit-ptr + 1;

    /* append the kms list to the end of the sit list */
    sit-ptr.next = head-kms-list-ptr;
    head-kms-list-ptr = 'null';
  end-if;
  else
    /* check to see where first node in kms list overlaps sit list, */
    /* i.e., if the node is a descendent of the current node        */
    found = 'false';

    /* set pointer to the head of the sit list */
    sit-ptr = head-sit-list-ptr;
    while (sit-ptr <> 'null' AND not found)
      if (sit-ptr.seg-name = head-kms-list-ptr.seg-name)
        found = TRUE;
      end-if;
      else
        sit-ptr = sit-ptr + 1;
    end-while;

    if (! found)
      print ("Error - match not found in GnOp");
    end-if;
    else
      /* found a valid overlap so set the parent pointer of the */
      /* first node of the kms list to the node above the node  */
      /* in the sit list that matched.                          */
      head-kms-list-ptr.parent = sit-ptr.prev;

      /* walk down to the end of the status list */
      status-ptr = first-status-node-ptr;
      while (status-ptr.next <> 'null')
        status-ptr = status-ptr + 1;
```

```
allocate a new status node:
append the status node to the status list:
kms-ptr = head-kms-list-ptr;
prev-kms-ptr = kms-ptr;
prev-sit-ptr = sit-ptr;
done = 'false';
while (not done)
  /* now we walk down the kms and sit lists to see how */
  /* much overlap the sit list contains                */
  free (kms-ptr.result-file);
  kms-ptr.result-file = sit-ptr.result-file:
  kms-ptr = kms-ptr + 1;
  sit-ptr = sit-ptr + 1;
  if (kms-ptr = 'null' OR sit-ptr = 'null')
    done = 'true';
  end-if;
  else
    /* both lists still contain nodes so increment them */
    prev-sit-ptr = sit-ptr;
    prev-kms-ptr = kms-ptr;
  end-else;
end-while;

if (kms-ptr = 'null')
  /* case where sit list contained all of kms list */
  status-ptr = prev-kms-ptr;
  status-ptr.status = MATCHALL;
end-if;
else
  /* case where sit list contained part of the kms list */
  status-ptr = kms-ptr;
  status-ptr.status = MATCHPART;
end-else;

/* now append kms list to the end of the sit list */
if (sit-ptr = 'null')
  prev-sit-ptr.next = head-kms-list-ptr;
end-if;
else
  while (sit-ptr.next <> 'null')
    sit-ptr = sit-ptr + 1:
  sit-ptr.next = head-kms-list-ptr;
end-else;
```

```
        head-kms-list-ptr = 'null';
      end-else;

    end-else;

  end-else:

end-else;

end-proc;

proc SPECIAL-RETRIEVE()
      /* When a GN or GNP operation has been selected without any segment   */
      /* search arguments specified, the normal GN or GNP operation of      */
      /* returning the next segment occurrance is skipped. Instead we       */
      /* consider this a special retrieve to return all segment occurrances */
      /* below the segment the currency pointer is pointing to.             */


 if (head-sit-list-ptr = 'null')
  print ("Error - status list null for SpecRetOp");
 end-if;
 else
  /* walk down to the end of the status list */
  status-ptr = first-status-node-ptr;
  while (status-ptr.next <> 'null')
   status-ptr = status-ptr + 1;

  allocate a new status node;
  status-ptr = head-kms-list-ptr;
  append status node to the status list;

  /* walk down to the end of the sit list - the last node */
  /* represents the current segment                       */
  sit-ptr = head-sit-list-ptr;
  while (sit-ptr.next <> 'null')
   sit-ptr = sit-ptr + 1;

  /* any nodes in the kms list with parent pointer = null    */
  /* must have the parent pointer set to the current segment */
  kms-ptr = head-kms-list-ptr:
  while (kms-ptr <> 'null')
   if (kms-ptr.parent = 'null')
    kms-ptr.parent = sit-ptr;
   end-if;
   kms-ptr = kms-ptr + 1;
  end-while;
```

```
    /* append the kms list to the end of the sit list */
    sit-ptr.next = head-kms-list-ptr;
    head-kms-list-ptr = 'null';
  end-else:

end-proc;

proc GET-NEXT-PARENT()
       /* A GNP operation is used to access the database just below the */
       /* current node the currency pointer is pointing to, rather than */
       /* having to specify the access path from the root as in a GU.   */


  if (head-kms-list-ptr.cmd-code <> 'StarF')
    perform GET-NEXT();
  end-if:
  else
    /* Once a valid GNP operation has been detected, we are sure the   */
    /* currency pointer is set to a segment (node) somewhere in the    */
    /* hierarchy with legitimate children beneath it. We then take the */
    /* parent pointer of the first node of the kms list and set it to  */
    /* the next to last node of the sit list. A status node is then    */
    /* created and set to point to the first node of the kms list.     */
    /* Finally, the kms list is appended to the sit list.              */

    /* walk down to the end of the sit list */
    sit-ptr = head-sit-list-ptr;
    while (sit-ptr.next <> 'null')
      sit-ptr = sit-ptr + 1;

    /* check to see if the head of the kms list is a valid child of the */
    /* next to last node of the sit list                                */
    if (head-kms-list-ptr.seg-name is not a valid child of the
                  next to last node in the sit list)
      print ("Error - valid child not found");
    end-if;
    else
      /* since it is a valid child, set the parent pointer of the first */
      /* node of the kms list to the next to last node of the sit list  */
      head-kms-list-ptr.parent = sit-ptr.prev;
```

```
      /* walk down to the end of the status list */
      status-ptr = first-status-node-ptr;
      while (status-ptr.next <> 'null')
      status-ptr = status-ptr + 1;

      allocate a new status node;
      status-ptr = head-kms-list-ptr;
      append the status node to the status list;

      /* append the kms list to the end of the sit list */
      sit-ptr.next = head-kms-list-ptr;
      head-kms-list-ptr = 'null';
    end-else;

  end-else;

end-proc;
```

```
dli-kc()

/*  This procedure accomplishes the following:    */
/*  (1) Checks Si-operation to determine whether  */
/*  we are creating a DB or querying the DB.      */
/*                                                */
/*  (2) Depending on the si-operation the cor-    */
/*  responding procedure is called.               */

{
 int c;

 initialize kc-curr-pos;
 initialize kc-ptr;

 switch (kc-curr-pos->Si-operation)
   {
    case CreateDB:
        load-tables();
        break;

    case ExecRetReq:
        requests-handler();
        break;

    default:
        printf("Error !!!!!!!");
        break;
   }
}


requests-handler()

/*  This procedure accomplishes the following:    */
/*  Calls dli-action until all DLI queries are    */
/*  processed.                                    */

{
 while (kc-curr-pos != NULL)
   {
    dli-action();
    get next Sit-info node to work on;
    set kc-curr-pos equal to this node;
   }
}
```

```
dli-action()
/*  This procedure accomplises the following:    */
/*  Uses a case statement based on the           */
/*  operation to determine the correct proc. to  */
/*  call.                                        */

{


  switch(kc-curr-pos->Si-operation)
    {
      case GuOp:
      case GnOp:
      case GnpOp:
          GU-proc();
          break;

      case GhuOp:
      case GhnOp:
      case GhnpOp:
          GHU-proc();
          break:

      case IsrtOp:
          GU-proc();
          break:

      case DletOp:
          Delete-proc(kc-curr-pos);
          printf("Dlet operation complete");
          break;

      case ReplOp:
          GU-proc();
          break;

      case SpecRetOp:
          Spec-ret-proc(kc-curr-pos);
          printf("SpecRet operation complete");
          break;

    }
  if (FAILURE)
    printf("Operation could not be completed due to ERROR !!!");
}
```

GHU-proc()

```
/*  This procedure accomplishes the following:    */
/*  Establishes a current position in the DB      */
/*  by calling dli-execute(), do-next-retrieve(), */
/*  and retract-a-level() so that the proper      */
/*  results can be returned.                      */

{
 char    *var-str-alloc();
 int     i;


 /* If the kc-curr-pos is also the di-fst-sit-pos, then
    we copy the Si-abdl-req over to Si-template (since
    it is fully formed) and then call dli-execute(). */
 if (kc-curr-pos == kc-ptr->di-fst-sit-pos->Ssi-req-pos)
  {
    i = strlen(kc-curr-pos->Si-abdl-req);
    allocate enough space for Si-template;
    strcpy(kc-curr-pos->Si-template,kc-curr-pos->Si-abdl-req);
    dli-execute():
    if (results-are-not-returned)
      retract-a-level();
  }

 /* Else this is a subsequent DLI query. Hence, we need to know
    where we are in the hierarchy of the DB.  The MATCH procedure
    will tell me this and I therefore use a flag it sets to base
    my next actions. */
 else
  {
    switch (kc-ptr->di-curr-sit-pos->Ssi-status)
      {
        /* This case is when the EOR of the last query is the
           same as the EOR of the current query. */
       case MATCHALL:
           /* If there is only 1 value in the buffer, then
              I need to get another value if there is one. */
           if (kc-curr-pos->Si-result-file->hfi-count <= 1)
             retract-a-level():
```

```
            /* Else I just move the buffer file pointer to
               the next value so that a new current position
               in the DB is established. */
            else
              {
                move file pointer;
              }
            break;

          /* This case is where the EOR of the current request does not
             match with the EOR of the previous query. */
          case MATCHPART:
            build-request();
            dli-execute();
            if (results-are-not-returned)
              retract-a-level();
            break;
        }
      }

  /* Until I hit the EOR or there is a failure, keep processing the
     abdl queries. */
  while ((kc-curr-pos->Si-EOR != TRUE) && (!FAILURE))
    {
      do-next-retrieve();
      if (results-are-not-returned)
        retract-a-level();
    }
  if (FAILURE)
    {
      return(FAILURE);
    }
}
```

GU-proc()

```
/*  This procedure accomplishes the following:    */
/*  Establishes a current position in the DB      */
/*  by calling dli-execute(), do-next-retrieve(), */
/*  and retract-a-level() so that the proper      */
/*  results can be returned. When the results are */
/*  returned, dli-kfs() is called so that they    */
/*  may be displayed.                             */

{
 char            *var-str-alloc();
 struct  Sit-info  *temp-ptr;


 /* If the kc-curr-pos is also the di-fst-sit-pos. then
    we copy the Si-abdl-req over to Si-template (since
    it is fully formed) and then call dli-execute(). */
 if (kc-curr-pos == kc-ptr->di-fst-sit-pos->Ssi-req-pos)
   {
    allocate enough space for Si-template:
    strcpy(kc-curr-pos->Si-template,kc-curr-pos->Si-abdl-req);
    dli-execute();
    if (results-are-not-returned)
      retract-a-level();
   }


 /* Else this is a subsequent DLI query. Hence, we need to know
    where we are in the hierarchy of the DB.  The MATCH procedure
    will tell me this and I therefore use a flag it sets to base
    my next actions. */
 else
   {
    switch (kc-ptr->di-curr-sit-pos->Ssi-status)
      {


       /* This case is when the EOR of the last query is the
          same as the EOR of the current query. */
       case MATCHALL:
           /* If there is only 1 value in the buffer, then
              I need to get another value if there is one. */
          if (kc-curr-pos->Si-result-file->hfi-buff-loc >
                    kc-curr-pos->Si-result-file->hfi-count)
            retract-a-level();
          else
            dli-kfs();
          break;
```

```
        /* This case is where the EOR of the current request does not
           match with the EOR of the previous query. */
        case MATCHPART:
            build-request();
            dli-execute();
            if (results-are-not-returned)
              retract-a-level();
            break;

      }
    }
 while ((kc-curr-pos->Si-EOR != TRUE) && (!FAILURE))
  {
    do-next-retrieve();
    if (results-are-not-returned)
      retract-a-level();
  }
 if (FAILURE)
  {
    return(FAILURE);
  }
 /* If the loop pointer is set, then we need to perform
    loop-handler(). */
 if (kc-curr-pos->Si-loop != NULL)
  {
    /* If the loop pointer is also the EOR, then all we do
       is empty this buffer of its results */
    if (kc-curr-pos->Si-loop->Si-EOR == TRUE)
      while (kc-curr-pos->Si-result-file->hfi-buff-loc <=
              kc-curr-pos->Si-result-file->hfi-count)
        dli-kfs();
    else
      {
        temp-ptr = kc-curr-pos->Si-loop->Si-next;
        while (temp-ptr != NULL)
          {
            temp-ptr->Si-result-file->hfi-status = RETRACTTIME;
            fclose(temp-ptr->Si-result-file->hfi-buff.fi-fid);
            temp-ptr = temp-ptr->Si-next;
          }
        loop-handler(kc-curr-pos->Si-loop->Si-next);
      }
  }

 if (FAILURE)
   return(FAILURE);
}
```

```
build-request()

/*  This procedure accomplishes the following:      */
/*  Builds an abdl request in the Si-template       */
/*  pointed to by the kc-curr-pos.  This procedure  */
/*  works from the back of the Si-abdl-req in       */
/*  building the request.                           */

{
 int            i,
                j,
                k,
                t,
                z;
 struct  Sit-info  *par-ptr,
         *or-ptr;
 char            c;
 int             firstime;

 i = j = string length of(kc-curr-pos->Si-abdl-req);
 par-ptr = kc-curr-pos->Si-parent;
 kc-curr-pos->Si-template = NULL;
 allocate enough space for kc-curr-pos->Si-template;
 firstime = TRUE;

 /* Working backwards in Si-abdl-req */
 while (i >= 0)
  {
   fill Si-template with contents of Si-abdl-req 'til an '*' is hit;

   /* If there is no value in the previous hfi-curr-buff-val,
      then one must be fethced so that the request can be built */
   if ((par-ptr->Si-result-file->hfi-curr-buff-val == NULL) &&
               (par-ptr != NULL))
    {
     z = i;
     /* Determine how large this value will be */
     z = i - z;
     using this value allocate space for it;
     fetch a value from the result buffer;
     put this value in hfi-curr-buff-val;
     if c == ' ', then need to read to EOL so that file ptr
        will be correct location when next value is fetched;
     place a ' ' at end of template;
    }
```

```
    if (i != 0)
      {
       put just obtained value into hfi-curr-buff-val;
       put hfi-curr-buff-val into Si-template;

       skip over the asteriks we just filled with a value;

       /* If the "or" flag is set TRUE by the KMS, then there are */
       /* places in the abdl-request where we should not move up  */
       /* the hierarchy to continue building the request.  In     */
       /* other words, we need to continue using the same value.  */
       if (kc-curr-pos->Si-or == TRUE)
         {
          if (firstime == TRUE)
            {
             firstime = FALSE;
             or-ptr = par-ptr;
            }
         A:while (an ASTERIK or 'r' has not been detected in Si-abdl-req)
            {
             fill Si-template with Si-abdl-req;
            }
          if (kc-curr-pos->Si-abdl-req[i] == 'r')
            {
             t = i;
             if (an 'o' followed by a ' ' is detected in Si-abdl-req)
               par-ptr = or-ptr;
             else
               {
                continue filling Si-template with Si-abdl-req;
                goto A;
               }
            }
          else
            par-ptr = par-ptr->Si-parent;
         }
       else
         par-ptr = par-ptr->Si-parent;
      }
  }

}
```

```
do-next-retrieve()

/*  This procedure accomplishes the following:    */
/*  (1) Sets the kc-curr-pos to the next SIT.     */
/*                                                */
/*  (2) Calls build-request() and then executes   */
/*  the complete abdl request.                    */

{
  kc-curr-pos = kc-curr-pos->Si-next:
  build-request():
  dli-execute();
}


dli-execute()

/*  This procedure accomplishes the following:    */
/*  (1) Sends the request to MBDS using           */
/*  TI-S$TrafUnit() which is defined in the Test  */
/*  Interface.                                    */
/*                                                */
/*  (2) Calls dli-chk-requests-left() to ensure   */
/*  that all requests have been received.         */

{

  TI-S$TrafUnit(kc-curr-pos->Si-template);
  dli-chk-requests-left():
```

```
dli-check-requests-left()

/*  This procedure accomplishes the following:    */
/*  (1) Receives the message from MBDS by calling */
/*  TI-R$Message() which is defined in the Test   */
/*  Interface.                                    */
/*                                                */
/*  (2) Gets the message type by calling          */
/*  TI-R$Type.                                    */
/*                                                */
/*  (3) If not all the responses to the request   */
/*  have been returned, a loop is entered. Within */
/*  this loop a case statement separates the      */
/*  responses received by message type..          */
/*                                                */
/*  (4) If the response contained no errors,      */
/*  then procedure TI-R$Req-res() is called to    */
/*  receive the response from MBDS.               */
/*                                                */
/*  (5) If no results are returned, then          */
/*  the boolean results-are-not-returned is set   */
/*  to TRUE.                                      */
/*                                                */
/*  (6) If the message contained an error,        */
/*  then procedure TI-R$ErrorMessage is called    */
/*  to get the error message and then procedure   */
/*  TI-ErrRes-output is called to output the      */
/*  error message.                                */


{
  int          msg-type,
               err-msg,
               done;
  char         *response;
  struct ReqID  rid;
  int          rid;

  results-are-not-returned = FALSE;
  done = FALSE;
```

```
while (!done)
 {
  TI-R$Message();
  msg-type = TI-R$Type();
  switch (msg-type)
   {
    case CH-ReqRes:
        done = TI-R$Reg-res(&rid,response);
        switch (kc-curr-pos->Si-operation)
         {
          case GuOp:
          case GnOp:
          case GnpOp:
              if (string length of(response) == 0)
               results-are-not-returned = TRUE;
              else
               if (End of Request == TRUE)
                {
                 file-results();
                 dli-kfs();
                }
               else
                file-results();
              break;

          case GhuOp:
          case GhnOp:
          case GhnpOp:
              if (string length of(response) == 0)
               results-are-not-returned = TRUE;
              else
               if (End of Request == TRUE)
                {
                 file-results();

                 printf("operation completed");
                }
               else
                file-results();
              break;

          case IsrtOp:
              if (End of Request == TRUE)
               printf("insert accomplished");
              else
               if (string length of(response) == 0)
                results-are-not-returned = TRUE;
               else
                file-results();
```

```
                break;
            case DletOp:
                if (string length of(response) == 0)
                  results-are-not-returned = TRUE;
                else
                  file-results();
                break;

            case SpecRetOp:
                if (string length of(response) == 0)
                  results-are-not-returned = TRUE;
                else
                  file-results();
                break;

            case ReplOp:
                if (End of Request == TRUE)
                  printf("replace accomplished");
                else
                  if (string length of(response) == 0)
                    results-are-not-returned = TRUE;
                  else
                    file-results();
                break;
          }
        break;

        case ReqsWithErr:
            /*Handle error conditions*/
            break;
      }/*end switch*/
    }/*end while*/
}/*end procedure*/
```

```
Delete-proc(x)
struct Sit-info *x;

/* This procedure accomplishes the following:    */
/* (1) Is called by dli-action and deletes a     */
/*     node and all its children.                */
/* (2) It works recursively by calling           */
/*     delete-setup(). delete-setup can in turn  */
/*     call Delete-proc. Hence, mutual recur-    */
/*     sion.                                     */

{

  kc-curr-pos = x;
  while (kc-curr-pos != NULL)
    {
          build-request();
          dli-execute();
        if (there is a child node)
          {
            kc-curr-pos = kc-curr-pos->Si-child;
            if (kc-curr-pos->Si-operation == GuOp)
              {
                build-request();
                dli-execute();
                if (results-are-not-returned)
                  break;
                else
                  {
                    kc-curr-pos = kc-curr-pos->Si-child;
                    delete-setup(kc-curr-pos);
                  }
              }
            else
              Delete-proc(kc-curr-pos);
          }
        if (there is a sibling node)
          {
            kc-curr-pos = kc-curr-pos->Si-sibling;
```

```
            if (kc-curr-pos->Si-operation == GuOp)
              {
                build-request();
                dli-execute();
                if (results-are-not-returned)
                  break;
                else
                  {
                    kc-curr-pos = kc-curr-pos->Si-child;
                    delete-setup(kc-curr-pos);
                  }
              }
            else
              Delete-proc(kc-curr-pos);
          }/*End of if*/
      }/*End of while*/
}/*End of procedure*/
```

```
delete-setup(x)
struct Sit-info *x;

/*  This procedure accomplishes the following:    */
/*  (1) Sets up a base node from which we base    */
/*      our recursion.                            */
/*  (2) Until we hit the end of its result-file   */
/*      we keep calling Delete-proc so that all   */
/*      appropriate nodes are deleted.            */

{
  int                   z;
  int                   *buff-loc,
                        *buff-count;
  struct   hie-file-info    *file-ptr;
  char                  c;

  kc-curr-pos = x;
  file-ptr = kc-curr-pos->Si-parent->Si-result-file;
  buff-loc = (file-ptr->hfi-buff-loc);
  buff-count = (file-ptr->hfi-count);
  fid = file-ptr->hfi-buff.fi-fid;

  while (buff-loc <= buff-count)
    {
      Delete-proc(x);
      skip over attribute name;
      z = 0;
      get the next value of the result file;
      buff-loc = buff-loc + 1;
    }
  kc-curr-pos = kc-curr-pos->Si-parent;
  clean-up-buffer();
}
```

```
Spec-ret-proc(x)
struct Sit-info *x;

/*  This procedure is called when it is necessary to */
/*  process special retrieves, i.e., those retrieves */
/*  where we have to retrieve a node's children as   */
/*  well.  Hence, this procedure is patterened after */
/*  the Delete-proc procedure, i.e., mutual recur-   */
/*  sion.                                            */

{
 int                    done;
 struct   hie-file-info   *file-ptr;
 int                    buff-loc,
                        buff-count;

 kc-curr-pos = x;
 file-ptr = kc-curr-pos->Si-result-file;
 buff-loc = (file-ptr->hfi-buff-loc);
 buff-count = (file-ptr->hfi-count);
 done = FALSE;

 while (!done)
  {
   build-request();
   dli-execute();
   if (results-are-not-returned == FALSE)
    if (there is a child node)
     spec-ret-setup(kc-curr-pos->Si-child);
   if (there is a sibling node)
    Spec-ret-proc(kc-curr-pos->Si-sibling);
   done = TRUE;
  }
 if (we are not at the end of the file)
  {
   close file;
   open file;
   buff-loc = 1;
   while (buff-loc <= buff-count)
    dli-kfs():
  }
}
```

```
spec-ret-setup(x)
struct Sit-info *x;

/* This procedure is similar to delete-setup in that */
/* it establishes a base node from which our recur-  */
/* sion is based.  Values are fetched from the base  */
/* node's result file until an EOF is determined.    */


{
 int                    buff-loc,
                        buff-count;
 struct  hie-file-info  *file-ptr;
 char                   c;
 int                    z;

 kc-curr-pos = x;
 file-ptr = kc-curr-pos->Si-parent->Si-result-file;
 buff-loc = (file-ptr->hfi-buff-loc);
 buff-count = (file-ptr->hfi-count);

 while (buff-loc <= buff-count)
  {
   Spec-ret-proc(x);
   skip over attribute name;
   z = 0:
   c = getc(fid);
   get the next value from the result file
   buff-loc = buff-loc + 1;
  }
 kc-curr-pos = kc-curr-pos->Si-parent;
 clean-up-buffer():
}
```

```
retract-a-level()

/*  This procedure accomplishes the following:    */
/*  Simulates retracting a level in the DB    */
/*  hierarchy.  This is done by using values in   */
/*  the previous SIT buffer to build requests     */
/*  until we either receive some results or we    */
/*  do not in which case we retract again.  The   */
/*  stopping condition for retracting is being    */
/*  at the BOR and its buffer is exhausted.       */

{
 int                 i;
 char                c;
 struct  hie-file-info  *curr-fptr,
   .                 *prev-fptr;
 int                 buff-loc;

 curr-fptr = kc-curr-pos->Si-result-file;
 prev-fptr = kc-curr-pos->Si-prev->Si-result-file;
 buff-loc = (prev-fptr->hfi-buff-loc);

 /* This is our stopping condition. */
 if ((kc-curr-pos is BOR) and
       (all elements in the result buffer have been used)
  {
   return(FAILURE = TRUE);
  }

 /* Else, we attempt to receive some results. */
 else
  {
   while (there are still results in the buffer to check)
    {
     pass over attribute name;
     i = 0;
     c = get a character from result file;
     load hfi-curr-buff-val with the attr value;
     build-request();
     dli-execute();
     buff-loc = buff-loc + 1;
     if (results-are-not-returned == FALSE)
      {
       return;
      }
    }
```

```
    if (we were unable to obtain any results)
      {
        kc-curr-pos = kc-curr-pos->Si-prev;
        clean-up-buffer();
        retract-a-level();
      }
   }
}


clean-up-buffer()

/* This procedure accomplishes the following:    */
/* (1) Sets hfi-status to RETRACTTIME.           */
/*                                               */
/* (2) Resets hfi-count to 0.                    */


{
struct   hie-file-info   *buff-ptr;
int                      *buff-count;
int                      *buff-loc;

  buff-ptr = kc-curr-pos->Si-result-file;
  buff-count = (buff-ptr->hfi-count);
  buff-loc = (buff-ptr->hfi-buff-loc);

  /* Set status to RETRACTTIME so that current results are overwritten */
  buff-ptr->hfi-status = RETRACTTIME;

  /* Reset buff-count and buff-loc to 0 */
  buff-count = 0;
  buff-loc = 0;

  buff-ptr->hfi-curr-buff-val = NULL;
  close kc-curr-pos file buffer;
}
```

```
init-buffer()

/*  This procedure accomplishes the following:    */
/*  (1) Copies the user's ID name into a temp     */
/*  string.                                       */
/*                                                */
/*  (2) Converts the current dbi-buff-count to    */
/*  a string.                                     */
/*                                                */
/*  (3) Increments the above count to reflect     */
/*  the fact that the next time this procedure    */
/*  is called it initialize a new buffer.         */
/*                                                */
/*  (4) strcat above count to temp.               */
/*                                                */
/*  (5) strcat BUFF-FILE-SUFFIX to temp.          */
/*                                                */
/*  (6) strcpy temp over to hfi-buff.fi-fname.    */

{
  char   temp[FNLength + 1];
  char   count[FNLength + 1];

  strcpy(temp,cuser-hie-ptr->ui-li-type.li-dli.di-curr-db.cdi-dbname);
  num-to-str(count,kc-ptr->di-buff-count);
  kc-ptr->di-buff-count = kc-ptr->di-buff-count + 1;
  strcat(temp,count);
  strcat(temp,BUFF-FILE-SUFFIX);
  strcpy(kc-curr-pos->Si-result-file->hfi-buff.fi-fname,temp);
}


load-tables()

/*  This procedure accomplishes the following:    */
/*  (1) Calls dbl-template which is already       */
/*  defined in the Test Interface.  It loads the  */
/*  template file.                                */
/*                                                */
/*  (2) Calls dbl-dir-tbls() also defined in      */
/*  the Test Interface.  It loads the descriptor  */
/*  files.                                        */
{
  struct  rtemp-definition  template;

  dbl-template(&template,kc-ptr->di-ddl-files->ddli-temp.fi-fid);
  dbl-dir-tbls(kc-ptr->di-ddl-files->ddli-desc.fi-fid);
}
```

```
loop-handler(x)
struct Sit-info *x;
/*  This procedure accomplishes the following:    */
/*  (1) Determines if the hfi-count is 1 or less. */
/*  If it is, then we need to get another value   */
/*  if there is one.                              */
/*                                                */
/*  (2) Else we just empty the buffer of the      */
/*  kc-curr-pos out.                              */



{
 struct   hie-file-info   *file-ptr;
 int                     *buff-loc,
                         *buff-count;
 char                     c;
 int                      z;


 kc-curr-pos = x;
 file-ptr = kc-curr-pos->Si-prev->Si-result-file;
 buff-loc = (file-ptr->hfi-buff-loc);
 buff-count = (file-ptr->hfi-count);

while (buff-loc <= buff-count)
 {
    loopit(kc-curr-pos);
    skip over attribute name;
    z = 0;
    get the next value from the result file;
    buff-loc = buff-loc + 1;
 }
 kc-curr-pos = kc-curr-pos->Si-prev;
 clean-up-buffer();
}
```

```
loopit(x)
struct Sit-info *x;
{
  kc-curr-pos = x;
  build-request();
  dli-execute();

  if (results-are-not-returned)
    return:
  else
    if ((kc-curr-pos->Si-next != NULL) &&
         (kc-curr-pos->Si-next->Si-BOR != TRUE))
      {
        kc-curr-pos->Si-next->Si-result-file->hfi-buff-loc = 1;
        loop-handler(kc-curr-pos->Si-next);
      }
  if (kc-curr-pos->Si-EOR == TRUE)
    while (kc-curr-pos->Si-result-file->hfi-buff-loc <=
            kc-curr-pos->Si-result-file->hfi-count)
      dli-kfs();
}


put-in-buff(instr)
  char *instr;

/*  This procedure accomplishes the following:   */
/*  Puts the incoming string form file-results   */
/*  into the correct file buffer.                */

{
  int i;

  for(i=0;instr[i] != EMARK;i++)
    putc(instr[i],kc-curr-pos->Si-result-file->hfi-buff.fi-fid);

  putc(' ',kc-curr-pos->Si-result-file->hfi-buff.fi-fid);
}
```

```
file-results()

/* This procedure accomplishes the following:    */
/* (1) Opens a file to place the results in.     */
/*                                               */
/* (2) Keeps track of how many results have      */
/* been received.                                */
/*                                               */
/* (3) Puts the results in their own line.       */

{
 char               *response,
                    *first-attr,
                    *temp-str;
 int                *num-values,
                    *buff-loc,
                    curr-pos,
                    res-len;
 struct  hie-file-info  *file-ptr;

 /* Next three statements are initialization */
 initialize file-ptr:
 initialize buff-loc;
 initialize num-values;

 /* If this is the first time then we open file for write status */
 if (file-ptr->hfi-status == FIRSTTIME)
   {
    init-buffer();
    open file for write mode;
    set hfi-status to RESTTIME;
    buff-loc = buff-loc + 1;
   }

 /* If hfi-status is RETRACTTIME, then must overwrite stuff in
    exiting file.  Thus, file is opened for write status. */
 else
   if (file-ptr->hfi-status == RETRACTTIME)
     {
      open file for write mode:
      set hfi-status to RESTTIME;
      buff-loc = buff-loc + 1;
     }
```

```
  /* If above two conditions don't hold, then just open file for
     append status. */
  else
    {
     open file for append status;
    }

response = kc-ptr->di-kfs-data.kfsi-hie.khi-response;
res-len = string length of(response);
curr-pos = 1;

    /* Read first attribute from response */
    read-dli-response(first-attr,curr-pos);

    /* Put this attribute in buffer */
    put-in-buff(first-attr);

    /* Read the value corresponding to this attribute */
    read-dli-response(temp-str,curr-pos);

    /* Put this value in the buffer */
    put-in-buff(temp-str);

    /* Increment the count of values */
    num-values = num-values + 1;

    /* While we are not at the end of the response */
    while (curr-pos < (res-len - 2))
      {
       read-dli-response(temp-str,curr-pos);

       /* If the attribute name just read in is not the same as the
          first attribute name previously read in, then we put it and
          its value on the same line in the buffer as the first attribute */
       if (strcmp(first-attr,temp-str) != 0)
         {
          put-in-buff(temp-str);
          read-dli-response(temp-str,curr-pos);
          put-in-buff(temp-str);
         }
```

```
        /* If they are the same, then we need to start a new line in
           the buffer. */
        else
          {
           put-in-buff("0);
           put-in-buff(temp-str);
           read-dli-response(temp-str,&curr-pos);
           put-in-buff(temp-str);
           num-values = num-values + 1:
          }
      }
  close file:
  open file:
}


read-dli-response(outstr,pos)
  char  *outstr:
  int   *pos:

/*  This procedure accomplishes the following:    */
/*  Reads the next value of the response buffer.  */

{
  int   i;
  char  *response:

  response = kc-ptr->di-kfs-data.kfsi-hie.khi-response:
  load outstr with the contents of response until an End Marker
     is detected:
  put a ' ' in outstr;
}
```

```
dli-kfs()
/*   This procedure accomplishes the following:  */
/*   Pulls a segment occurence from the proper   */
/*   buffer and displays it to the user.         */
{
  char c;

  pull a value from kc-curr-pos file buffer;
  print this value;
  printf("0);
  buff-loc = buff-loc + 1;
}
```

APPENDIX E - THE DL/I USERS' MANUAL

A. OVERVIEW

The DL/I language interface allows the user to input transactions from either a file or the terminal. A transaction may take the form of either database descriptions of a new database, or DL/I requests against an existing database. Database descriptions may only be input from a file, while DL/I requests may be input from either a file or the terminal. The DL/I language interface is menu-driven. When the transactions are read from either a file or the terminal, they are stored in the interface. If the transactions are database descriptions, they are executed automatically by the system. If the transactions are DL/I requests, the user is prompted by another menu to selectively choose an individual DL/I request to be processed. The menus provide an easy and efficient way to allow the user to view and select the methods in which to process DL/I transactions. Each menu is tied to its predecessor, so that by exiting each menu the user is moved up the "menu tree". This allows the user to perform multiple tasks in a single session.

B. USING THE SYSTEM

There are two operations the user may perform. The user may either define a new database or process requests against an existing database. The first menu displayed prompts the

user for an operation to perform. This menu, hereafter referred to as MENU1, looks like the following:

```
Enter type of operation desired
     (l) - load a new database
     (p) - process old database
     (x) - return to the operating system

ACTION ----> _
```

Upon selecting the desired operation, the user is prompted to enter the name of the database to be used. When loading a new database, the database name provided may not presently exist in the database schema. Likewise, when processing requests against an existing database, the database name provided has to exist in the present database schema. In either case, if an error occurs, the user is told to rekey a different name. The session continues once a valid name is entered.

If the "p" operation is selected from MENU1, a second menu is displayed that asks for the mode of input. This input may come from a data file or interactively from the terminal. This generic menu, MENU2, looks like the following:

```
Enter mode of input desired
     (f) - read in a group of transactions from a file
     (t) - read in transactions from the terminal
     (x) - return to the previous menu

ACTION ----> _
```

If users wish to read transactions from a file, they are prompted to provide the name of the file that contains those transactions. If users wish to enter transactions directly from the terminal, a message is displayed reminding them of the correct format and special characters that are to be used.

If the "1" operation is selected from MENU1, a second menu is displayed that is identical to MENU2 except that the "t" option is omitted. Since the transaction list stores both database descriptions and DL/I requests, two different access methods have to be employed to send the two types of transactions to the KMS. Therefore, our discussion branches to handle the two processes the user may encounter.

1. Processing Database Descriptions (DBDs)

When the user has specified the filename of DBDs, further user intervention is not required. It does not make sense to process only a single DBD out of a set of DBDs that produce a new database, since they all have to be processed at once and in a specific order. Therefore, the mode of input is limited to files, and the transaction list of DBDs is automatically executed by the system. Since all the DBDs have to be sent at once to form a new database, control should not return to MENU2 where further transactions may be input. Instead, control returns to MENU1 where the user may select a new operation or a new database to process against.

2. Processing DL/I Requests

In this case, after users have specified the mode of input, they conduct an interactive session with the system. First, all DL/I requests are listed to the screen. As the DL/I requests are listed from the transaction list, a number is assigned to each DL/I request in ascending order starting with the number one. The number is printed on the screen beside the first line of each DL/I request. Next, an access menu, called MENU3, is displayed which looks like the following:

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding DL/I requests
     (d)   - redisplay the list of DL/I requests
     (r)   - reset the currency pointer to the root
     (x)   - return to the previous menu

ACTION ----> _
```

One selection from MENU3 needs further explanation. The "r" selection causes the currency pointer to be repositioned to the root of the hierarchical schema so that subsequent requests may access the complete database. Examples of the need to utilize this option are: before executing DL/I GUs or ISRTs.

Since the displayed DL/I requests may exceed the vertical height of the screen, only a full screen of DL/I requests are displayed at one time. If the desired DL/I request is not displayed on the current page, the user may

depress the RETURN key to display the next page of DL/I requests. If the user only desires to display a certain number of lines, after the first page is displayed the user may enter a number, and only that many lines of DL/I requests are displayed. If users are only looking for certain DL/I requests, once they have found them, they do not have to page through the entire transaction list. By depressing the "q" key, control is broken from listing DL/I requests, and MENU3 is displayed. Under normal conditions, when the end of the transaction list has been viewed, MENU3 appears.

Since DL/I requests are independent items, the order in which they are processed does not matter. The users have the choice of executing however many DL/I requests they desire. A loop causes the transaction list and MENU3 to be redisplayed after each DL/I request has been executed so that further choices may be made. Unlike processing DBDs, control returns to MENU2 since the user may have more than one file of DL/I requests against a particular database, or the user may wish to input some extra DL/I requests directly from the terminal. Once the user is finished processing on this particular database, the user may exit back to MENU1 to either change operations or exit to the operating system.

C. DATA FORMAT

When reading transactions from a file or the terminal, there has to be some way of distinguishing when one transaction ends and the next begins. Transactions are allowed to span multiple lines, as evidenced by a typical multi-level DL/I GU followed by a GN. This example also shows that our definition of transaction incorporates one or more requests. This allows a group of logically related requests to be executed as a group. When a transaction contains multiple requests, each request has to be separated by an end-of-request flag. In our system this flag is the "!" character. Since the system is reading the input line by line, an end-of-transaction flag is required. In our system this flag is the "@" character. Likewise, the system needs to know when the end of the input stream has been reached. In our system the end-of-file flag is represented by the "$" character. The following is an example of an input stream with the necessary flags that are required when multiple transactions are entered:

```
TRANSACTION #1
@
TRANSACTION #2
REQUEST #1
!
REQUEST #2
!
.
:

!
REQUEST #n
@
TRANSACTION #3
@
.
:

@
TRANSACTION #n
$
```

D. RESULTS

When the results of the executed transactions are sent back to the user's screen, they are displayed exactly the same way individual DL/I requests are displayed (see Section B.2). The following consolidates the user's options:

KEY	FUNCTION
return	Displays next screenful of output
(number)	Displays only (number) lines of output
q	Stops output, MENU1 is then redisplayed

LIST OF REFERENCES

1. Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," in the Proceedings of the New Directions in Computing Conference, Trondheim, Norway, August, 1985; also in Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

2. Banerjee, J., Hsiao, D. K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, March 1980.

3. Weishar, D. J., The Design and Analysis of a Complete Hierarchical Interface for a Multi-Backend Database System, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

4. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970, also in Corrigenda, Vol 13., No. 4, April 1970.

5. Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," Communications of the ACM, September 1971.

6. Rothnie, J. B. Jr., "Attribute Based File Organization in a Paged Memory Environment," Communications of the ACM, Vol. 17, No. 2, February 1974.

7. The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-77-7, DBC Software Requirements for Supporting Relational Databases, by J. Banerjee and D. K. Hsiao, November 1977.

8. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-85-002, A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Gains, by S. A. Demurjian, D. K. Hsiao and J. Menon, February 1985.

9. IBM Corporation, Information Management System/Virtual Storage Application Programming Reference Manual, IBM Form No. SH20-9026.

10. Boehm, B. W., Software Engineering Economics, Prentice-Hall, 1981.

11. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-84-012, Software Engineering Techniques for Large-Scale Database Systems as Applied to the Implementation of a Multi-Backend Database System, by Ali Orooji, Douglas Kerr and Daivid K. Hsiao, August 1984.

12. The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-82-1, The Implementation of a Multi Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS, by D. S. Kerr et al, January 1982.

13. Kernighan, B. W., and Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

14. Howden, W. E., "Reliability of the Path Analysis and Testing Strategy," IEEE Transactions on Software Engineering, Vol. SE-2, September 1976.

15. Johnson, S. C., Yacc: Yet Another Compiler-Compiler, Bell Laboratories, Murray Hill, New Jersey, July 1978.

16. Lesk, M. E. and Schmidt, E., Lex - A Lexical Analyzer Generator, Bell Laboratories, Murray Hill, New Jersey, July 1978.

17. Date, C. J., An Introduction to Database Systems, 3d ed., Addison Wesley, 1982.

18. Shienbrood, E., More - A File Persual Filter for CRT Viewing, Bell Laboratories, Murray Hill, New Jersey, July 1978.

19. Kloepping, G. R., and Mack, J. F., The Design and Implementation of a Relational Interface for the Multi-Lingual Database System M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5100	1
5.	Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	1
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
7.	Timothy P. Benson P. O. Box 1974 Woodbridge, Virginia 22193	3
8.	Gary L. Wentz 111 Appian Way Pasadena, Maryland 21122	3
9.	Gary R. Kloepping Route 1, Box 99 Santa Rosa, Texas 78593	2
10.	John F. Mack 2934 Emory Street Columbus, Georgia 31903	2

214035

Thesis
B3916 Benson
c.1 The design and implementation of a hierarchical interface for the multi-lingual database system.